VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2019

Bc. Patrik Švikruha

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

VYUŽITÍ TECHNOLOGIE BLAZOR S FRAMEWORKEM DOTVVM

USING BLAZOR TECHNOLOGY WITH THE DOTVVM FRAMEWORK

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

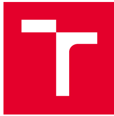
AUTOR PRÁCE Bc. Patrik Švikruha

AUTHOR

VEDOUCÍ PRÁCE doc. Ing. Ivo Lattenberg, Ph.D.

SUPERVISOR

BRNO 2019



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**

Ústav telekomunikací

Student: Bc. Patrik Švikruha **ID:** 164423

Ročník: 2 ***Akademický rok:*** 2018/19

NÁZEV TÉMATU:

Využití technologie Blazor s frameworkem DotVVM

POKYNY PRO VYPRACOVÁNÍ:

Prostudujte webový standard WebAssembly v ASP.NET Core a experimentální technologii Blazor. Pokuste se využít tyto technologie pro zlepšení frameworku DotVVM, který je k dispozici jako open-source.
Cílem je použití této technologie na minimalizaci PostBacků ze strany klienta na server v open-source MVVM webovém frameworku DotVVM. Řešení by mělo umožňovat záložní možnost pro případ, že webový prohlížeč nebude technologii WebAssembly podporovat.

DOPORUČENÁ LITERATURA:

[1] MAREŠ, Amadeo. 1001 tipů a triků pro C# 2010. Brno: Computer Press, 2011. ISBN 978-80-251-3250-0.

[2] MACDONALD, Matthew, Adam FREEMAN a Mario SZPUSZTA. ASP.NET 4 a C# 2010: tvorba dynamických stránek profesionálně. Brno: Zoner Press, 2011. Encyklopedie Zoner Press. ISBN 978-80-7413-145-5.

Termín zadání: 1.2.2019 ***Termín odevzdání:*** 16.5.2019

Vedoucí práce: doc. Ing. Ivo Lattenberg, Ph.D.
Konzultant:

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

UPOZORNĚNÍ:
Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Diplomová práca sa zaoberá využitím technológií Blazor a WebAssembly s frameworkom DotVVM. Práca obsahuje fundamentálne informácie a teoretický rozbor webových technológií a princípov. Tieto technológie sú prerekurzormi technológii WebAssembly, Blazor a DotVVM. Práca sa detailne zameriava na WebAssembly, ktorá umožňuje diverzifikáciu na poli klientských webových technológií. V tejto práci sa využíva časť technológie Blazor, konkrétne CLI runtime Mono, ktorý je skompilovaný do WebAssembly formátu a tým umožňuje beh IL kódu v prehliadači. Práca sa zameriava na minimalizáciu počtu PostBackov medzi klientom a serverom, ktorá je možná vďaka vyššie spomínanému Mono runtime.

KĽÚČOVÉ SLOVÁ

DotVVM, WebAssembly, WASM, Blazor, ASP.NET Core, .NET Core, .NET, Mono, JavaScript, JavaScript engine, LLVM, AOT kompilátor, JIT kompilátor, WSL

ABSTRACT

Diploma thesis deals with usage of Blazor and WebAssembly with DotVVM framework. Thesis contains fundamental information about web technologies and priciples. These technologies are precursors of technologies like WebAssembly Blazor and DotVVM. Diploma thesis focuses on WebAssembly, that enables diversification on client side web technologies. In this work is used just part of Blazor, specifically CLI runtime Mono, which is compiled to WebAssembly. This compiled runtime enables execution of IL code in browser. Thesis is focused on minimalization of PostBacks betweeen client and server, with usage of CLI Mono runtime.

KEYWORDS

DotVVM, WebAssembly, WASM, Blazor, ASP.NET Core, .NET Core, .NET, Mono, JavaScript, JavaScript engine, LLVM, AOT compiler, JIT compiler, WSL

ŠVIKRUHA, Patrik. *Využití technologie Blazor s frameworkem DotVVM*. Brno, 2019, 97 s. Diplomová práca. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedúci práce: doc. Ing. Ivo Lattenberg, Ph.D.

VYHLÁSENIE

Vyhlasujem, že som svoju diplomovú prácu na tému „Využití technologie Blazor s frameworkem DotVVM“ vypracoval samostatne pod vedením vedúceho diplomovej práce, využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej diplomovej práce ďalej vyhlasujem, že v súvislosti s vytvorením tejto diplomovej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona Českej republiky č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákoníka Českej republiky č. 40/2009 Sb.

Brno ....

podpis autora

POĎAKOVANIE

Rád by som poďakoval Tomášovi Hercegovi a Adamovi Ježovi za odborné pripomienky k mojej práci. Veľká vďaka za odborné rady, konzultácie a hodnotné pripomienky k mojej práci patrí môjmu vedúcemu doc. Ing. Ivanovi Lattenbergovi Ph.D.. V neposlednom rade ďakujem mojej priateľke Petre za trpezlivosť a morálnu podporu.

Brno . . . .

podpis autora

Obsah

Úvod 12

Slovník pojmov 13

I Teoretická časť 15

1 História World Wide Web-u 16
1.1 CERN a World Wide Web . . . 16
1.1.1 Prvá webová stránka . . . 17
1.2 Začiatky vývoja webových stránok . . . 17
1.3 Expanzia vývoja webových stránok . . . 18
1.4 Netscape . . . 19
1.4.1 Vojny prehliadačov (Browser wars) . . . 21

2 Webový vývoj 23
2.1 Špecifiká webového vývoja . . . 23
2.1.1 Webové prostredie . . . 23
2.1.2 HTTP . . . 24
2.1.3 AJAX . . . 25
2.1.4 Model View Controller (MVC) architektúra . . . 27
2.2 Serverové technológie . . . 29
2.2.1 Common Gateway Interface (CGI) . . . 29
2.2.2 Skriptovacie (interpretované) serverové jazyky . . . 30
2.2.3 ASP.NET a ASP.NET Core . . . 30
2.2.4 Zhrnutie serverových technológii . . . 31
2.3 Klientské webové technológie . . . 31
2.3.1 Webový prehliadač . . . 34
2.3.2 HyperText Markup Language (HTML) . . . 36
2.3.3 Cascading style sheets (CSS) . . . 36
2.3.4 Javascript . . . 37

3 WebAssembly 43
3.1 História a dôvody vzniku . . . 44
3.1.1 Native Client, Portable Native Client . . . 46
3.1.2 asm.js . . . 47
3.1.3 Zhrnutie Native Client a asm.js . . . 48
3.1.4 LLVM . . . 48

3.2 Princíp fungovania WebAssembly . . . 49
3.3 Porovnanie WebAssembly a JavaScriptu . . . 51
3.4 Štandardy W3C a podpora prehliadačov . . . 53
3.5 Aplikácie ktoré používajú WebAssembly . . . 53
3.6 Budúcnosť a rozpracovaná funkcionalita . . . 54

4 .NET - Common Language Infrastructure 57
4.1 Common Language Infrastructure . . . 57
4.2 .NET Framework . . . 58
4.3 .NET Core . . . 58
4.4 Mono . . . 58
4.5 .NET Standard . . . 59
4.6 .NET a WebAssembly . . . 60

5 Blazor 61
5.1 Špecifikácia Blazor . . . 61
5.1.1 Blazor - klientsky prístup . . . 62
5.2 Aktuálny stav Blazoru . . . 64

6 DotVVM Framework 65
6.1 MVVM . . . 65
6.2 MVVM a DotVVM . . . 66
6.2.1 Základné koncepty . . . 66
6.2.2 Serverová časť . . . 67
6.2.3 Klientská časť (prehliadač) . . . 68

II Praktická časť 70

7 Prerekvizity 71
7.1 Aktivácia WSL . . . 71
7.2 Inštalácia WSL . . . 72
7.2.1 Emscripten SDK . . . 73
7.3 Voliteľné prerekvizity pre OS Windows . . . 73
7.3.1 .NET Core Sdk . . . 74
7.3.2 Node.js . . . 74
7.3.3 Visual Studio . . . 74

8 Implemetácia 76
8.1 Konceptuálne riešenie zadania . . . 76
8.2 Použité knižnice a tooling . . . 77

8.3 Štruktúra výsledného riešenia 77
8.3.1 DiplomaDemo.NetStandard 77
8.3.2 DiplomaDemo.DotVVM 79
8.3.3 Bootstraper 81

9 Záver **82**

Literatúra **83**

Zoznam používaných skratiek **93**

Zoznam príloh **95**

A Obsah prílohy **96**

Zoznam obrázkov

1.1 Tim Berners-Lee s jeho prehliadačom WorldWideWeb 17
1.2 Logo prehliadača Netscape Navigator 19
2.1 Schéma fungovania klasickej webovej aplikácie 24
2.2 Schéma fungovania aplikácie s AJAX 26
2.3 Principiálna schéma MVC (Model 2) architektonického vzoru 28
2.4 Zjednodušená schéma architektúry prehliadača 34
2.5 Oficiálne logo JavaScriptu 38
2.6 Principiálna schéma fungovania JavaScriptu 42
3.1 Oficiálne logo WebAssembly 43
3.2 Princíp fungovania LLVM toolchainu 49
3.3 Zjednodušená schéma zásobníkového počítača - Stack machine 49
3.4 Principiálna schéma fungovania WebAssembly 50
3.5 Porovnanie vykonávania JavaScriptového súboru a WASM súboru . . 52
4.1 Principiálna schéma fungovania .NET platformy 57
4.2 Principiálna schéma AOT kompilácie 59
4.3 Grafické znázornenie .NET Standardu 60
5.1 Principiálna schéma kompilácie kódu - JIT prístup. 62
5.2 Principiálna schéma kompilácie kódu - AOT prístup. 63
6.1 Principiálna schéma MVVM architektonického vzoru 65
6.2 Principiálna schéma fungovania DotVVM na serveri 67
6.3 Principiálna schéma fungovania DotVVM v prehliadači 68
7.1 Rozšírenie DotVMM vo Visual Studiu 75
8.1 Principiálna schéma integrácie DotVVM a mono.wasm 76
8.2 Ukážka demo aplikácie 81

Zoznam tabuliek

1.1 Prehľad verzii prehliadača Netscape 20
2.1 Prehľad Browser Engines 35
2.2 Verzie HTML 36
2.3 Verzie CSS 37
3.1 Štandardy W3C pre WebAssembly 53

Zoznam výpisov kódov

2.1 [HTML5] Ukážka kódu HTML5 a CSS 37
2.2 [JS] Ukážka kódu JavaScriptu 39
2.3 [JS] Vytvorenie plne optimalizovaných polí 41
2.4 [JS] Vytvorenie čiastočne optimalizovaného poľa 41
3.1 [JS] Explicitne definované typy premenných 45
3.2 [JS] Kód pred optimalizáciou 47
3.3 [JS] Kód po optimalizácii 47
3.4 [WAT] Webassembly textová forma 50
3.5 [C] Program v jazyku C pred kompiláciou 51
3.6 [JS] Zavolanie funkcie main() pomocou JavaScript interoop 51
7.1 [PowerShell] Zistenie verzie buildu OS Windows 71
7.2 [PowerShell] Aktivácia WSL v OS Windows 71
7.3 [PowerShell] Stiahnutie a inštalácia Ubuntu 18.04 71
7.4 [Shell] Stiahnutie a inštalácia .NET závislostí 72
7.5 [Shell] Stiahnutie a inštalácia Emscripten SDK závislostí 72
7.6 [Shell] Stiahnutie a inštalácia Python a PowerShell 72
7.7 [Shell] Naklonovanie repozitára s Emscripten SDK 73
7.8 [PowerShell] Inštalácia .NET Core SDK pomocou choco 74
7.9 [PowerShell] Inštalácia Node.js pomocou choco 74
7.10 [PowerShell] Inštalácia Visual Studio a workloads pomocou choco . . 74
8.1 [PowerShell] Pridanie nuget balíčkov pomocou dotnet CLI 78
8.2 [C#] Ukážka pridania závislostí z DotvvmStartup.cs 80

Úvod

V mojej diplomovej práci sa budem zaoberať webovým štandardom WebAssembly a technológiou Blazor. Jedná sa o pomerne nové technológie, v prípade WebAssembly sa stále pracuje na webovom štandarde a technológia je pod intenzívnym vývojom. WebAssembly je zatiaľ hotová ako Minimal Valuable Product (MVP).

V prípade technológie Blazor sa zmenilo označenie dňa 18. apríla 2019 označenie **experimental** na oficiálne podporovaný **preview**. Technológia Blazor je zaujímavá alternatíva k dnešným SPA frameworkom, a tiež dobrý pokus ako priblížiť .NET ekosystém a tooling na stranu klienta, do prehliadača.

V teoretickej časti sa budem zameriavať na uvedenie historických súvislostí, ktoré majú priamy vplyv na vývoj vyššie spomínaných technológií. V úvodnej kapitole chcem tiež poukázať na to, prečo je dôležité, aby bola každá nová technológia štandardizovaná vyššou autoritou (napr. W3C) a zároveň bol jej zdrojový kód opensource.

Na čo by som chcel upozorniť je aktuálnosť informácii v tejto práci, ktorú nemôžem zaručiť, keďže v prípade WebAssembly sa jedná o intenzívne vyvíjaný štandard a v prípade Blazor sa jedná o preview verziu.

Moja práca nesie v názve technológie Blazor a DotVVM, čo v prípade technológie Blazor môže byť mierne zavádzajúce. Z technológie Blazor budem používať iba časť, konkrétne CLI runtime Mono (mono.wasm) bez rendrovacieho engine. Taktiež budem využívať ešte stále experimentálnu AOT kompiláciu C# do WebAssembly.

V praktickej časti bude vytvorený konceptuálny návrh a demo aplikácia, ktorá bude používať DotVVM a CLI runtime Mono skompilovaný do WebAssembly. Demo bude demonštrovať zdieľanie kódu medzi serverom a klientom, čo umožní minimalizáciu PostBackov.

Slovník pojmov

V mojej práci budem používať anglický tvar určitej softvérovej terminológie, pretože niektoré preklady neexistujú, alebo by mohli byť, podľa môjho názoru, pre čitateľa zavádzajúce.

Nasledujúce pojmy preto nebudem prekladať a budem ich základ používať v anglickom tvare:

- **Toolchain** - jedná sa o súbor programovacích nástrojov, ktoré sa používajú na komplexnejšiu postupnosť úloh pri vytváraní softvéru.
- **Tooling** - program, alebo sada programov ktoré zameriavajú na uľahčenie vývoja softvéru.
- **Runtime** - je to program, ktorý prekladá určitú formu medzikódu do strojového kódu architektúry (x86, AMD64, ARM). Jeden z možných prekladov je behové prostredie.
- **Sandbox** - jedná sa o bezpečné prostredie, ktoré je oddelené od hosťovského systému alebo ostatných systémov. Toto oddelenie zabezpečuje určitý stupeň ochrany.
- **Engine** - tento pojem sa používa ako mentálny model pre označenie komplexného softvéru skladajúceho sa z rôznych technológii.
- **Front-end ...** - pojem slúži na označenie technológií, ktoré súvisia s užívateľským rozhraním (UI)
- **Back-end ...** - pojem slúži na označenie technológií, ktoré nesúvisia s UI.
- **Full-stack ...** - pojem slúži na zjednotenie front-end a back-end technológií.
- **Bootstrap** - význam je niečo medzi inštaláciou a nastavením. Používa sa pre označenie niečoho, čo pripravuje a samo spúšťa určitý softvér.
- **Compilation target** - jedná sa o platformu, do ktorej sa kompiluje určitý kód. Medzi najznámejšie compilation targets patria architektúry x86, AMD64 a ARM.
- **Portable compilation target** - compilation target ktorý je nezávislí na architektúre na ktorej bude bežať výsledný kód (napr. .NET, Java, WebAssebmly).
- **JIT (Just In Time)** - pojem spojený s kompiláciou kódu. Pojem JIT kompilátor znamená, že medzikód je kompilovaný pri jeho použití.
- **AOT (Ahead Of Time)** - pojem spojený s kompiláciou kódu. Pojem AOT kompilácia znamená, že celý kód je skompilovaný na compilation target „hneď“.
- **Overhead** - čas ktorý je nutný na nepriame vykonanie určitej úlohy (jedná sa hlavne o čas prípravy). Pre príklad uvediem spustenie aplikácie ktorá je napísaná v .NET-e. Pred samotným spustením efektívneho kódu aplikaćie je nutnú určitý čas (overhead) na inicializáciu .NET runtime.
- **Codebase** - kolekcia zdrojových súborov ktorá je použitá pre určitý druh

softvéru.

- **Compile-time** - čas kedy sa vykonáva kompilácia.
- **Debbuging** - proces pri ktorom sa hľadajú a odstraňujú z kódu chyby.

Časť I

Teoretická časť

1 História World Wide Web-u

Nasledujúca kapitola bude popisovať historické súvislosti, ktoré majú priamy vplyv na vývoj technológií na webe. Úlohou tejto kapitoly je dať čitateľovi všeobecný prehľad o technológiách a príčinách ktoré formovali World Wide Web (skrátene web).

1.1 CERN a World Wide Web

V roku 1989 Tim Berners-Lee, softvérový inžinier v *Conseil Européen pour la recherche nucléaire* (CERN) v Ženeve, uviedol koncept World Wide Web-u (Webu). O 2 roky neskôr sa táto myšlienka stala skutočnosťou vďaka uvedeniu prvej webovej stránky. Tak vznikol dnes všadeprítomný internet.

Koncept webu vznikol ako návrh riešenia problému so zdieľaním výsledkov experimentov medzi jednotlivými laboratórnymi pracoviskami, univerzitami a výskumnými inštitúciami. Ak prišli vedci do CERN-u za účelom výskumu, neexistoval štandardizovaný spôsob ako si odniesť výsledky experimentov do svojho materského laboratória alebo ich jednoducho zdieľať celej vedeckej obci. Jeden z najväčších problémov bola preprava dát z CERN-u do domovskej inštitúcie. Znamenalo to prepravu dát na fyzických médiach a s tým spojené riziko poškodenia fyzického média. Ďalším problémom bola nekonzistentnosť duplikovaných, pri úprave vypracovaných záverov, z čoho vyplývali rozdielne závery.

Dokument, ktorým Tim Berners-Lee myšlienku Webu (World Wide Web-u), predložil v roku 1989 vedeniu CERNu. Tento návrh však nebol hneď schválený a prešiel viacerými úpravami. [67][2]

V októbri 1990 boli špecifikované 3 základné technológie pre WWW:

- **HyperText Markup Language (HTML)** - formát, ktorý okrem štrukturálnej reprezentácie dokumentov umožňuje aj "prelinkovanie"na iné dokumenty alebo súbory. Patrí do rodiny Standard Generalized Markup Language (SGML).
- **Unified Document Identifier (UDI)** - unikátna adresa každého dokumentu. Neskôr sa táto adresa rozdelila na Unified Resource Indentifier (URI) a Uniform Resource Locator (URL).
- **HyperText Transport Protocol (HTTP)** - je jednoduchý textový protokol, ktorý umožňuje komunikáciu na základe modelu klient-server (požiadavka-odpoveď)[1].

[1]Protokol bude popísaný podrobnejšie v kapitole 2.1.2.

1.1.1 Prvá webová stránka

V tom istom roku, ako bola vydaná špecifikácia pre technológie webu, napísal Tim Berners-Lee prvý prehliadač s názvom **WorldWideWeb**, a tiež prvý webový server s názvom **httpd**.

Týmto vytvoril prvú webovú stránku, ktorá bežala v sieti CERN-u, a ktorá bola sprístupnená vedeckej obci a verejnosti bezplatne. Hlavná myšlienka internetu, a to slobodný prístup k informáciám, bola zrealizovaná a položili sa základy internetu, ktorý poznáme dnes.

V roku 1994 Tim Berners-Lee založil konzorcium W3C. Konzorcium schvaľuje webové štandardy a jeho členovia sú okrem firiem s najväčším podielom na internetovom trhu (napr. Amazon, Google Inc., IBM Corporation, Microsoft Corporation) aj individuálni weboví vývojári.[95] Toto konzorcium vydáva štandardy pre HTML, Cascading Style Sheets (CSS) a WebAssembly a ďalšie webové technológie.[45][3]

Obr. 1.1: *Tim Berners-Lee s jeho prehliadačom WorldWideWeb (zdroj:* `https://home.cern/science/computing/birth-web`*)*

1.2 Začiatky vývoja webových stránok

Pred rokom 1993 neexistoval web server v takom zmysle, v akom ho poznáme dnes. V tejto dobe bola hlavnou súčasťou web serveru implementácia HTTP protokolu. Neoddeliteľnou súčasťou webového prehliadača bol web server, ktorý umožňoval zdieľanie lokálnych HTML dokumenty.

Prvé stránky taktiež neobsahovali grafické prvky a skôr sa podobali rozhraniu terminálu. Prvá webová stránka, ktorú vytvoril Tim Berners-Lee, existuje na webovej adrese `http://info.cern.ch/` a má byť pripomienkou príbehu začiatkov World Wide Web-u.[44]

V tej dobe neexistovala žiadna technológia, ktorá by umožňovala určitú dynamickosť HTML dokumentov - webových stránok. Stránky boli iba statické. Pre jednu URI adresu, sa všetkým oživovateľom zobrazil rovnaký obsah. Tiež je nutné poznamenať, že stránky v tej dobe neobsahovali grafické prvky ako tlačidlá, textové polia a podobne. Jednalo sa iba o štrukturované dokumenty s naformátovaným textom a odkazy na ostatné dokumenty, prípadne súbory.

Prvou možnosťou vytvorenia interakcie s užívateľom bola Common Gateway Interface (CGI). CGI umožňovalo dynamicky poskladať stránku na základe požiadavky užívateľa. Interakcia na strane klienta - v prehliadači, nebola možná, a JavaScript ako ho poznáme dnes bol hudbou budúcnosti.

Jedným z prvých prehliadačov, ktorý sa podobal dnešným webovým prehliadačom, bol Mosaic. Tento prehliadač podporoval ako prvý `img` element, ktorý umožnil na stránke zobraziť obrázok. Vytvoreniu tohto HTML elementu predchádzal návrh od pána Marc Andreesena. Prehliadač Mosaic priblížim v kapitole 1.4 [1][3]

1.3 Expanzia vývoja webových stránok

Roky 1994 a 1995 znamenali expanziu webových stránok, pretože sa začali o web zaujímať obchodné spoločnosti, ktoré v ňom videli veľký zákaznícky potenciál pre svoje služby. Z webových stránok, ktoré obsahovali iba statické prvky, vznikli vďaka skriptovaciemu jazyku JavaScript[2] prvé webové aplikácie, ktoré umožňovali užívateľom interaktivitu.

V roku 1994 predstavila firma Netscape protokol Secure Sockets Layer (SSL) vo verzii 1. Táto verzia protokolou nebola nikdy verejne vydaná kvôli bezpečnostným dieram, avšak počas februára 1995 už Netscape vydal verziu 2.0. Nová verzia protokolu umožnila bezpečnú komunikáciu prostredníctvom HTTP.[74] Príchod šifrovanej komunikácie cez SSL umožnil vznik obchodných firiem ako Amazon, eBay, atď. Príchod týchto spoločností pomohol naštartovať vývoj webových aplikácii a web serverov. Tieto spoločnosti sponzorovali konzorcium W3C a doteraz sa podieľajú na štandardizácii webovej platformy.

Prvé web servery na základe HTTP požiadavkov spúšťali podprogramy, ktoré boli zväčša napísané v jazyku C. Webový server spúšťal podprogramy s argumentami ktoré získal z HTTP požiadaviek. Tieto podprogramy vytvorili na základe

[2]Bližšie bude popísaný v kapitole 2.3.4

vstupných argumentov daný HTML dokument. Tento spôsob vybavovania HTTP požiadaviek vznikol už v roku 1993 v National Center for Supercomputing Applications (NCSA), čo je americká federálna inštitúcia pre vývoj kyber-infraštruktúry a vývoj superpočítačov.[79]

1.4 Netscape

Veľmi dôležitou súčasťou histórie je aj proprietárny webový prehliadač Netscape. Jedná sa skôr o sériu webových prehliadačov, medzi ktoré patria Netscape Communicator, Netscape 6, Netscape, Netscape Browser a Netscape Navigator.

Obr. 1.2: *Svetoznáme logo prehliadača Netscape Navigator*

Tento prehliadač vznikol už v roku 1994 od autorov prehliadača Mosaic, čo bol prvý prehliadač ktorý umožňoval zobrazovanie obrázkov. Netscape mal vstavanú podporu pre protokoly FTP, telnet atď. Interný názov tohto prehliadača bol Mozilla, čo bola parafráza na „Mosaic killer“. Neskôr bol tento názov použitý na vytvorenie open-source softvérovej skupiny Mozilla Corporation, ktorá je dnes neodmysliteľnou progresívnou súčasťou webovej komunity. Vedúci tímu, ktorý tvoril prehliadač Mosaic, Marc Andreessen, v roku 1994 spolu s James H. Clarkom založili firmu Mosaic Communications Corporation. Táto firma začala vyvíjať prehliadač s interným názvom Mosaic Netscape. NCSA, ktorá bola autorom a vlastníkom práv k prehliadaču Mosaic, podala právnu výzvu o použití ochrannej známky Mosaic na firmu Mosaic Communications Corporation. Kvôli tejto výzve sa firma premenovala na Netscape Communications Corporation a prehliadač dostal pomenovanie Netscape Navigator.

Týmto vzniklo kultové meno Netscape, ktoré bolo synonymom inovatívneho prístupu na niekoľko ďalších rokov. Netscape už v prvej beta verzii obsahoval pokročilú funkcionalitu, vďaka čomu sa stal takmer okamžite vedúcim prehliadačom na trhu a vytlačil tak z tejto pozície Mosaic. Vo verzii 2 bol pridaný mailový klient a z Netscape sa stal tzv. „internet suite“. Takto sa označovali aplikácie, ktorá slúži ako webový prehliadač, emailový klient, download manager a editor HTML dokumentov. Verzia 3 pokračovala v úspechu verzií 1 a 2. Na trhu sa objavil Internet Explorer ktorý znamenal vážnu konkurenciu pre Netscape.

Verzia 4 priniesla rozdelenie na dva samostatné produkty. Netscape Navigator pokračoval v myšlienke internet suite a novovzniknutý Netscape Communicator obsahoval iba prehliadač a download manager.

Netscape stále viac pociťoval konkurenciu od Internet Exploreru, ktorý bol dodávaný k operačnému systému Microsoft Windows 95 zdarma. Verzia 4.0 zaznamenala komerčný úspech, aj napriek tomu že jádro prehliadača sa oproti verzii 3.* zmenilo iba minimálne.

Netscape Navigator a Communicator začal byť príliš komplexný softvér a z toho začal prameniť jeho úpadok. Spoločnosť Netscape Communications Corporation bola odkúpená spoločnosťou America Online (AOL) v roku 1998. Spoločnosť AOL zmenila obchodný model a začala ponúkať Netscape zadarmo. Ďalším vývojom bola poverená spoločnosť Mozilla Corporation. Cyklus vydávania nových verzii Netscape Communicatoru a Netscape Navigator bol stále dlhší. Touto skutočnosťou získal Internet Explorer 5 s lepšou podporou pre W3C štandardy HTML 4, CSS, DOM a ECMAScript, prvé miesto na trhu. Verzia 5 Netscape Navigator nebola nikdy verejne vydaná, pretože mala problémy s rendrovacím jadrom.

Mozilla dostala za úlohu od AOL napísať nové rendrovacie jadro, ktoré sa volalo **Gecko**. Mozilla vyvíjala okrem toho vlastný prehliadač ktorý ale ešte nebol pripravený na verejné vydanie. Kvôli tlaku z AOL bola Mozilla Corporation nútený vydať novú verziu Netscape Navigátoru. Kvôli problémom so starým rendrovacím jadrom sa rozhodli použiť jadro Gecko, ktoré však bolo ešte pod intenzívnym vývojom a nebolo pripravené na verejné vydanie.

Komunita tiež dosť ostro odsudzovala integráciu komunikačného nástroja Instant Message do Netscape Communicator-u, ktorú si vynútil AOL.

Tab. 1.1: *Prehľad verzii prehliadača Netscape*

Oficiálny názov	**Rok vydania**	**Verzie**	**Render engine**
Netscape Navigator	1994	1.0 → 3.0	Netscape
Netscape Communicator	1997	4.0 → 4.8	Netscape
Netscape 6	1998	6.x	Gecko 0.6 → 0.9.4.1
Netscape 7	2002	7.x	Gecko 1.0.1 → 1.7.2
Netscape Browser	2005	8.x	Gecko 0.9.3 → 1.5.0.11
Netscape Navigator	2007	9.x	Gecko 2.0.0.4 → 2.0.0.12

Tieto všetky problémy spôsobili to že komunita používajúca Netscape Navigátor si našla alternatívne prehliadače, povačšine sa jednalo o Internet Explorer, ktorý sa stal dominantným prehliadačom na trhu. Osudu Nescape Navigator-u nepomohlo ani to že v roku 2007 materská firma AOL prepustila vývojárov Netscape-u a vývoj produktu využila firmy Mercurial Communications. V roku 2008 následne AOL

oznámila, že nemá v pláne pokračovať vo vývoji Netscape kvôli nízkemu podielu na trhu.[42][25][104].

Tabuľka 1.1 zahŕňa všetky dôležité majoritné verzie Netscape-u, spolu s typom render engine[3] a oficiálnym názvom prehliadača.[104]

1.4.1 Vojny prehliadačov (Browser wars)

V období verzie Netscape 3 eskalovala prvá tzv. „browser war", čiže vojna prehliadačov, kedy sa o miesto na trhu bili dvaja najväčší konkurenti a to Netscape a Internet Explorer. Netscape zo začiatku vyhrával, ale tým že bol Internet Explorer dodávaný zadarmo k operačnému systému Microsoft Windows 95 a od verzie 4 mal lepšiu podporu webových štandardov W3C. Netscape tak od roku 1997 postupne strácal svoju pozíciu na trhu.

Prvú vojnu prehliadačov teda vyhral Internet Explorer a na 11 rokov sa stal najpoužívanejším webovým prehliadačom na svete. Internet Explorer, ktorý získal náskok pred Netscape-om vďaka tomu, že implementoval dohodnuté webové štandardy W3C a ako prvý implementoval CSS, spravil tú istú chybu ako Netscape na začiatku svojho zániku. Svojou implementáciou funkcií predbiehal štandardy W3C, a implementoval vlastnú funkcionalitu a kvôli architektonickým chybám sa stal zraniteľným voči malwarom a vírusom. Vývoj nových verzii Internet Exploreru sa postupne spomaľoval natoľko, že medzi okmi 2002 a 2006 vyšla iba 1 major verzia. Spomalenie cyklu vydávania nových verzií spôsobilo, že Internet Explorer nestíhal implementovať štandardy W3C dostatočne skoro. Spomalenie cyklu malo za následok aj to, že Microsoft nedokázal dodávať bezpečnostné záplaty dostatočne rýchlo. Tieto skutočnosti spôsobili že užívatelia postupne migrovali na konkurenčné prehliadače ako Mozilla Firefox alebo Opera.

V roku 2008 bol predstavený Google Chrome, ktorý má momentálne najväčší podiel na trhu. Túto pozíciu získal kombináciou vlastnosti medzi ktoré patria:

- 6 týždenný cyklus vydávania nových verzii[4]
- rýchla implementácia štandardov W3C
- n-násobné zrýchlenie JavaScriptu[5] oproti konkurenčným prehliadačom
- pre každý tab prehliadača sa vytvoril nový proces [6]

V roku 2015 predstavila firma Microsoft prehliadač Edge, nástupcu Internet Exploreru. Microsoft po predstavení nového prehliadača oznámil koniec vývoja prehliadača Internet Exploreru, čím tento prehliadač odsúdil podobne ako to AOL urobilo s Netscape.

[3]Render engine bude vysvetlený v kapitole 2.3.1

[4]Verzia k 27.4 je 76

[5]Jedna z základných webových technológii, ktorú bližšie predstavím v kapitole 2.3.4

[6]Konkurenčné prehliadače používali pre taby vlákna

Druhá bitka prehliadačov trvala od roku 2004 do 2018 a podľa slov Andreasa Gala, bývalého CTO Mozilla Corporation, síce bitku vyhral Google Chrome, ale stále nevyhral vojnu o Web.[34][6][38]

2 Webový vývoj

Webový vývoj sa od svojho počiatku do dnešnej podoby výrazne zmenil. Na začiatku sa používali statické stránky, ktoré iba zobrazovali existujúce dokumenty a klientská interakcia neexistovala. Tak ako sa internet rozširoval, rástli aj požiadavky na webové stránky, z ktorých sa po pridaní interaktívnych prvkov stali webové aplikácie. Nasledujúce kapitoly budú rozoberať technológie ktoré umožnili vznik klientskej interakcie a umožnili webu stať sa najrozšírenejšou aplikačnou platformou.

2.1 Špecifiká webového vývoja

Statické stránky neboli pre narastajúcu webovú komunitu dostatočné a preto začali vznikať rôzne prístupy ako dostať na web určitú dynamickosť webových stránok. Jedným z prvých úspešných krokov bola technológia CGI (Common Gateway Interface), ktorá umožňovala dynamicky vytvárať HTML dokumenty.

Expanziou internetu sa zvyšovali nároky na interakciu webových stránkok. Zvyšovanie nárokov viedlo k expanzii vzniku nových technológii na strane klienta (prehliadača) a na strane servera.

Webová platforma je charakteristická množstvom implementácii rovnakej technológie, ktorá je špecifikovaná štandardom. Nasledujúci zoznam obsahuje hlavné štandardizačné organizácie spolu s hlavnými štandardami:

- *World Wide Web Consortium* (**W3C**) - štandardizuje HTML, CSS a DOM
- *Ecma International* (**ECMA**) - štandardizuje ECMAScriptu.
- *Internet Engineering Task Force* (**IETF**) - Štandardizuje s pomocou W3C HTTP protokol pomocou RFC publikácii.
- *The Web Hypertext Application Technology Working Group* (**WHATWG**) - pracuje na JavaScript API pre DOM[1].

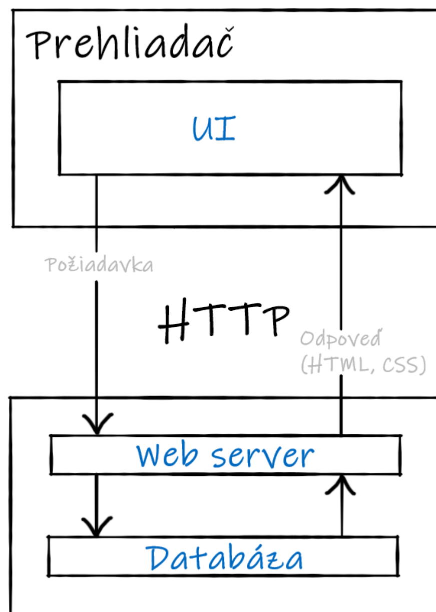
Nasledujúce kapitoly zhŕňajú teoretické princípy fungovania webových aplikácii. Táto teória je dôležitou pre-rekvizitou pred kapitolami o serverových (2.2) a klientských technológiach (2.3).

2.1.1 Webové prostredie

Webové prostredie je bezstavové prostredie založené na sieťovej architektúre klient-server. Táto architektúra funguje tak, že klient posiela HTTP požiadavky na ktoré server odosiela HTTP odpovede. Z tohto princípu teda server nevie o klientovi, až pokiaľ ho klient nekontaktuje a teda jediná možnosť komunikácie serveru a klienta

[1] Pretože špecifikácia ECMAScriptu nehovorí o JavaScriptovom API pre prácu s DOM a špecifikácia DOM je jazykovo agnostická.

je skrz požiadavku klienta na ktorú server posiela odpoveď.[2] Špecifická vlastnosť tejto architektúry je teda závislosť klienta na dátach, ktoré mu poskytuje server. Na obrázku 2.1 je znázornená schéma fungovania webovej aplikácie s výmenou dát medzi serverom a klientom.



Obr. 2.1: *Schéma fungovania klasickej webovej aplikácie*

Pôvodná myšlienka, kde server „iba“ poskytoval dáta ktoré klient zobrazoval HTML dokument, bola prekonaná vďaka klientským skriptovacím jazykom, ako napríklad JavaScript, alebo VBScript a na klientskú stranu sa začala implementovať pokročilejšia logika aplikácie, ktorá umožňovala interakciu aplikácie s užívateľom. Rozmach tohoto prístupu umožnilo predovšetkým pridanie podpory Dynamic HTML do špecifikácie DOM, ktoré podrobnejšie popíšem v kapitole 2.3.1. Logika a kód aplikácie je teda rozdelený na 2 hlavné časti a to serverovú a klientskú časť, kde každá plní svoju špecifickú úlohu. Toto rozdelenie vnieslo do vývoja ďalšiu komplexitu, ktorú množstvo webových aplikácii rieši pomocou MVC architektúru.

2.1.2 HTTP

Protokol HTTP je bezstavový protokol ktorý je základom pre komunikáciu na webe. Pracuje na princípe klient-server, kde klient posiela HTTP požiadavku na server a ten odpovedá HTTP odpoveďou. Existujú 4 verzie protokolu: HTTP/0.9, HTTP/1.0, HTTP/1.1 a HTTP/2.0.

Protokol vo verzií 0.9 obsahoval iba metódu GET a v odpovedi bol hypertext[3].

[2] Ak neberieme do úvahy WebSocket
[3] Obsah HTML dokumentu

Verzie protokolou 0.9 a 1.0 používali pre každý HTTP požiadavok nové TCP spojenie. Tento prístup bol veľmi neefektívny, pretože každé nové TCP spojenie spôsobovalo pri naviazaní overhead. HTTP/1.1 prišlo s myšlienkou zdielať jedno TCP spojenie pre viacero HTTP požiadaviek. V praxi to zaznamenávalo že sa vytvorila tzv. pipe HTTP požiadaviek, ktoré boli **blokujúce**. Táto pipe požiadaviek fungovala ako zásobník, čiže druhá HTTP požiadavka sa dostala na radu až po obslúžení prvej HTTP požiadavky.

Tento prístup znamenal že bolo výhodné mať vo webovej aplikácii čo najmenej súborov[4]. Súbory rovnakého typu (*.js, *.css) sa pomocou techniky pomenovanej **bundling** spojili do jedného súboru, ktorý sa nazýva **bundle**.

HTTP/2 priniesla **Multiple Request**, ktorý dokáže skrz jedno TCP spojenie poslať paralelne viacero HTTP požiadaviek na server. Tento prístup znamená to, že je výhodnejšie mať viacero malých súborov. Podľa webového portálu `https://caniuse.com/#feat=http2` podporuje HTTP/2 štandard viac ako 88% zariadení je preto vhodnejšie webové aplikácie optimalizovať pre použitie s HTTP/2 ktorý zároveň

HTTP/2 priniesla viac praktických vylepšení ktoré umožnili webovým aplikáciam rýchlejšie načítanie.

- Multiplexing (Multiple Request) - viacero HTTP požiadaviek skrz jedno TCP spojenie.
- Server Push - ak si pomocou HTTP požiadavky požiadam napríklad o HTML súbor, server vie že budem po prijatí HTML súboru žiadať o JavaScriptové a CSS súbory a preto tieto súbory pošle spolu s HTML dokumentom. Klient má samozrejme možnosť tieto súbory odmietnuť.
- Binárny protokol - miesto textovej reprezentácie príkazov sa používa ich binárna reprezentácia. Binárna reprezentácia znamená jednoduchšiu implementáciu HTTP na strane servera a klienta. Je tiež odolnejšia voči chybám (textová forma musí riešiť escapovanie a encoding riadiacich znakov) Je tiež menší overhead pri parsovaní dát a tým väčšia efektivita využitia sieťových prostriedkov.

Špecifikácia HTTP/2 **nevyžaduje** šifrované spojenie (s TLS), ale v praxi ho všetky prehliadače pre HTTP/2 vyžadujú.[35][32][15][43]

2.1.3 AJAX

AJAX je skratka pre Asynchronous JavaScript and XML, čo znamená súhrn pre vyššie spomínané techniky ako vytvoriť stránky, ktoré bežia na klientovi a zo serveru si doťahajú iba dáta, bez nutnosti znova načítať danú stránku. Táto skratka síce

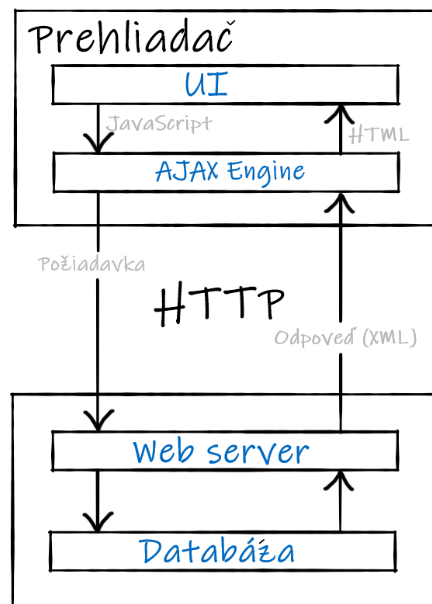
[4]Menej súborov = menej HTTP požiadaviek

hovorí o XML, ale v praxi sa dnes používa takmer výhrade JSON[5] formát. XML sa používal iba na začiatku pretože prehliadače neimplementovali vhodné API pre prácu s JSON.

Asi najznámejší zástupca aplikácií, ktoré využívajú AJAX, sú tzv. Single Page Apllication (SPA). Tieto aplikácie môžu udržovať stav, napríklad s použitím Web Storage, čo je úložisko na strane prehliadača. Na obrázkoch 2.1 a 2.2 je graficky znázornený rozdiel medzi „klasickou" stránkou a stránkou ktorá využíva AJAX.

Ako som spomínal vyššie, server môže na základe klientského AJAX požiadavku vracať rôzne formáty, ako napríklad XML, JSON alebo aj kus HTML, ktoré sa potom pomocou klientského Javascriptu vložia do DOM, čo má za následok prekreslenie HTML dokumentu.

Túto techniku využívajú Javascriptové frameworky ako napríklad React, ktorý pracuje s tzv. Virtual DOM, čo je Javascriptový objekt predstavujúci celý HTML dokument.



Obr. 2.2: *Schéma fungovania aplikácie s AJAX*

Pojem AJAX uviedol po prvý krát Jesse James Garrett-a v článku s názvom „*AJAX: A New Approach to Web Applications*" v roku 2005. Myšlienka dynamicky meniť obsah HTML dokumentu je však staršia, a bola uvedená v prehliadači Internet Explorer ako element **iframe**. V skratke sa jedná o stránku v stránke a v niektorých prípadoch sa tento prístup hodí, má však jendu obrovskú nevýhodu a to takú, že nie je možné pracovať naraz s DOM-om stránky z elementu **iframe** a DOM-om stránky na ktorej sa nachádza **iframe**.

[5]JavaScript Object Notation bude popísaný v kapitole 2.3.4

Na obrázku 2.2 je znázornená principiálna schéma fungovania AJAX. Požiadavka ide skrz AJAX engine, ktorý sa správa ako proxy medzi UI vrstvou a serverom. AJAX engine po obdržaní HTTP odpovede vo formáte XML (Json) upraví UI.[36][94]

2.1.4 Model View Controller (MVC) architektúra

Architektúry sa v softvérovom vývoji používajú preto, pretože umožňujú štandardizovať prístupy k riešeniu daného problému a tým uľahčiť a zefektívniť prácu na danom projekte. Projekt ktorý používa architektúru je lepšie testovateľný a umožňuje separáciu zodpovednosti medzi jednotlivé vrstvy. Týmto rozdelením má projekt udržateľnejší vývoj.

Pôvodnú verziu architektúry MVC navrhol pán Trygve Reenskaug pre Smalltalk už v roku 1979 a bola určená pre desktopové aplikácie. Táto MVC architektúra pracovala s hardvérovým oddelením jednotlivých komponentov počítača. Vstupné zariadenie bola klávesnica, ktorú reprezentoval Controller. Model bol dátové úložisko a View predstavoval zobrazovacie zariadenie - displej. V dostupnej literatúre a dokumentáciách rôznych frameworkov sa používa pre MVC viacero podobných definícií, kde sa MVC označuje buď ako architektúra, alebo ako prezentačný vzor (presentation pattern).

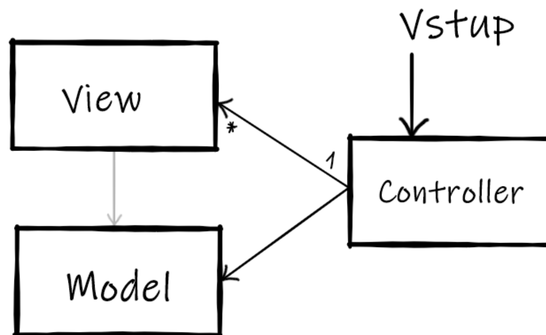
Komplexnosť UI sa zvyšovala a tým sa zvyšovala aj komplexnosť prezentačnej vrstvy. Toto jej zvýšenie malo za následok vznik rôznych modifikácii MVC architektúry. Tieto modifikácie založené na MVC obsahujú zväčša 3 časti, a to View, Model a Controller. Definície týchto modifikácii často kombinujú pôvodnú MVC architektúru s definíciou **Model 2** architektúry.

Určité modifikácie tohto architektonického vzoru boli definované s vlastným názvom, aby sa predišlo zámene s existujúcimi modifikáciami. Pre príklad spomeniem najznámejšie Model View Presenter a Model View ViewModel.

V nasledujúcom texte sa budem odkazovať na MVC architektúru Model 2. Túto architektúru využívajú webové aplikačné frameworky ako ASP.NET MVC, Ruby on Rails alebo Java Servlet.

View je UI reprezentácia dát. Model sú dáta ktoré zobrazuje View. Controller je mozog, ktorý obsahuje aplikačnú logiku a stará sa o vytváranie View a Modelu.

Vstup do architektúry MVC je prostredníctvom Controllera. Controller prijíma všetky požiadavky (vstupy) a obsahuje logiku pre ich spracovanie. Na základe tejto logiky vytvorí Model, ktorý obsahuje dáta. Controller sa tiež stará o inicializáciu View, ktorému poskytuje Model s dátami. Controller môže na základe vstupných argumentov požiadavky poskytovať rôzne View. Tento vzťah je znázornený na obrázku 2.3 pomocou 1 –> * notácie.



Obr. 2.3: *Principiálna schéma MVC (Model 2) architektonického vzoru*

Architektúra MVC sa používa v množstve dnešných moderných frameworkov a bola rozšírená hlavne vďaka ASP.NET a Java Servlet technológiám.

V praxi sa často používa „serverový“ MVC v kombinácii s “klientským“ MVC, čiže aplikácia môže používať MVC architektúru na 2 úrovniach. V obidvoch prípadoch však Controller predstavuje mozog celej architektúry a je zodpovedný za spracovanie vstupov. [92][76][11][47][48]

Klientský MVC

- View => Je to HTML dokument ktorý rendruje prehliadač a vidí ho užívateľ.
- Controller => Je zodpovedný za načítanie dát pre View a ich transformáciu do DOM[6]. Je napísaný v JavaScripte.
- Model => Je chápaný ako dáta, alebo zdroj dát (webový server).

Serverový MVC

- View => Je to súbor ktorý sa vracia ako odpoveď na klientsky HTTP požiadavok = môže to byť súbor, prípadne iba HTTP odpoveď.
- Controller => Je zodpovedný za spracovanie klientského HTTP požiadavku a vytvorenie následnej HTTP odpovede.
- Model => Je chápaný ako dáta.

Praktický príklad použitia architektúry

V praxi môže táto architektúra fungovať nasledovne (Príklad používa AJAX):

Užívateľ zadá do prehliadača url webovej aplikácie. Prehliadač vyšle požiadavku GET a webový server mu vráti HTML dokument. Prehliadač pri načítaní HTML dokumentu pošle ďalšie GET požiadavky na JavaScriptové súbory ktoré sú potrebné

[6]Reprezentácia HTML dokumentu, ku ktorej je možno pristupovať z JavaScriptu, bližšie bude popísaný v kapitole 2.3.1

pre beh webovej aplikácie. Po načítaní HTML dokumentu a všetkých skriptov sa inicializuje z JavaScriptu klientsky controller. Tento Controller vyrobí HTTP požiadavku (napríklad POST) a pošle ju na webový server. Webový server teraz funguje ako proxy a presmeruje túto požiadavku na servrový Controller ktorý ju spracuje a vráti HTTP POST odpoveď s JSON súborom ktorý obsahuje dáta pre webovú aplikáciu. Klientsky Controller túto odpoveď spracuje a upraví pomocou DOM API HTML dokument a Render Engine prehliadača sa postará o prerendrovanie danej webovej aplikácie pre užívateľa.

2.2 Serverové technológie

Požiadavky na servery v dnešnej dobe sú diametrálne odlišné od tých, ktoré boli pri počiatkoch webových aplikácii. Keď ešte nebol internet bežnou súčasťou života ako je dnes, požiadavky na webové aplikácie boli relatívne nízke. Webový server sa používal iba ako ako zdroj HTML dokumentov ktoré vracal na základe URI a smerovacej tabuľky k súborom ktorú mal pre HTTP požiadavky. So zvyšovaním požiadavkou v klientskú interakciu začal byť tento prístup nedostačujúci a bolo potrebné na základe určitých parametrov modifikovať HTML dokumenty ktoré tento webový server poskytoval užívateľom - prehliadačom.

Táto etapa je tiež špecifická tým že neexistoval štandardizovaný spôsob ako vytvárať, alebo modifikovať HTML dokumenty na strane webového serveru. Tento problém ako prvé riešilo Common Gateway Interface, a webové aplikácie začínali dostávať určité interaktívne prvky.

2.2.1 Common Gateway Interface (CGI)

Tento spôsob vybavovania klientských požiadavkov vznikol síce už v roku 1993, ale oficiálny názov Common Gateway Interface so štandardom vznikol až roku 1996, kedy bol vydaný pod RFC 3875 pre CGI verziu 1.1.

CGI umožňovalo na HTTP požiadavku odoslať HTML dokument vygenerovaný na základe určitých parametrov. Pri každej HTTP požiadavke spúšťal webový server ëxecutable"skript, lepšie povedané program, čiže nový proces, ktorý sa zavolal s danými parametrami a ktorý vrátil HTML dokument, alebo súbor. Prvé CGI skripty sa písali v jazykoch ako C, Perl, alebo dokonca .batch.

CGI teda znamenalo jeden proces pre vybavenie jednej HTTP požiadavky, čo znamenalo, že počet procesov na serveri vzrastal lineárne s počtom klientských HTTP požiadaviek, ktoré mal server obslúžiť. Požiadavky na servere pri tomto spôsobe obsluhy klientských požiadavkov boli pri vyššom počte návštevníkov webovej aplikácie obrovské, a tak sa hľadalo iné riešenie.

CGI je míľnik, ktorý sme prekonali a je dôležitou súčasťou histórie. CGI umožňoval totiž spracovanie klientskej HTTP požiadavky na serveri, čo je prelomový koncept, ktorý sa používa dodnes, ak si uvedomíme že na začiatku fungovali webové stránky iba tak, že sme dokázali na základe HTTP požiadavky iba vrátiť HTML dokument (na základe URI).[33]

2.2.2 Skriptovacie (interpretované) serverové jazyky

CGI teda umožňovalo jazykovo agnostický prístup k tomu, ako vytvárať dynamické webové stránky. Po združení niekoľko CGI programov ktoré boli napísané v jazyku C vytvoril v roku 1994 Rasmus Lerdorf skriptovací jazyk PHP (Personal Home Page/Hypertext Preprocessor). pre operačný systém Windows a webový server IIS vznikol v roku 1996 jazyk ASP (Active Server Pages), ktorý umožňoval prístup k Component Object Model (COM), čím sprístupnil funkcionalitu kompilovaných knižníc (napríklad .dll).[82][24]

PHP a ASP znamenali revolúciu, pretože umožňovali písať kusy kódu (skriptov) do HTML dokumentu, a tým tak „zjednodušiť“ prácu webovým developerom. Podstatný rozdiel oproti „bežnému“ CGI skriptu bol ten, že sa nepísal program ktorý vracia vytvorený HTML dokument, ale písal sa HTML dokument do ktorého sa písali kusy funkčné kódu - skripty, ktoré podľa potreby napríklad prevolávali databázu.

V praxi však tento prístup často znamenalo písanie tzv. špagety kódu, pretože ASP a PHP umožňovali písať do deklaratívneho HTML dokumentu funkčný kód, ktorý aktívne menil obsah dokumentu. Týmto vznikali dlhé a neprehľadné bloky kódu, ktoré bolo veľmi ťažké debuggovať a udržovať.

Ďalší dôležitý framework ktorý spopularizoval architektúru MVC, spomínanú v kapitole 2.1.4, sa nazýva Ruby On Rails. Tento framework, spolu s PHP a ASP sú priamymi potomkovia filozofickej línie CGI skriptov a umožnili rozvoj webových aplikácii.

2.2.3 ASP.NET a ASP.NET Core

Z technológie ASP vzniklo ASP.NET ktorý z dynamického skriptovacieho ASP spravil po pridaní .NET runtime [7] staticky typovanú technológiu, ktorá umožňovala písať stránky v C# alebo v VB.NET.

ASP.NET prinieslo Web Forms, ktoré umožnili vyvíjať webové aplikácie desktopovým štýlom a tým sa pomerne úspešne vyhnúť písaniu JavaScriptu (Web Forms ho generovali). Na technológiu Web Forms sa znieslo pomerne veľa kritiky, kvôli

[7] Bližšie budem .NET popisovať v kapitole 2.2.3

tomu že zakrývala prirodzené bezstavové prostredie webu a preto bolo mnoho webových aplikácii napísaných pomerne nekvalitne. Web Forms zakrývalo bezstavové prostredie webu pomocou konceptu ViewState ktorý udržoval stav webovej aplikácie.

ASP.NET Core je moderná reimplementovaná verzia ASP.NET, ktoré používa namiesto .NET Frameworku open source multiplatformovú implementáciu .NET Core. Vo svojej práci budem používať framework DotVVM[8], ktorý je založený na ASP.NET a ASP.NET Core.[40][65]

2.2.4 Zhrnutie serverových technológii

Z jednoduchých CGI rozhraní, sme sa prepracovali až k webovým aplikačným frameworkom[9], ktoré zjednodušujú vývoj a udržovanie webových aplikácii.

Pôvodná myšlienka CGI, čiže poskladanie HTML dokumentu na serveri je dodnes používaný a overený koncept. Vývoj dnešných webových aplikácii sa ale mení a skôr sa preferuje prístup, kde sa využíva výkon zariadenia kde beží daná webová aplikácia. Uvedením JIT pre JavaScript engine[10] a stále výkonnejšími zariadeniami na ktorých bežia prehliadače otáčajú prístup vývoja webových aplikácii. Serverová časť aplikácie sa používa iba ako zdroj statických HTML dokumentov, JS skriptov, CSS štýlov a ako nositeľ biznis logiky. Trend vývoja webových aplikácii sa teda sústredí na kompletné prenesenie aplikačnej logiky do prehliadača, na stranu klienta. Úlohy webových aplikačných frameworkov na strane servera sú teda hlavne rýchle obslúženie HTTP požiadavkov a malý footprint v hosťovskom operačnom systéme[11]. [78][58]

2.3 Klientské webové technológie

Webové aplikácie ktoré umožňujú klientskú interakciu sa stali štandardom a v dnešnej dobe je trend presúvať stále väčšie množstvo aplikačnej záťaže do klientskeho prehliadača (z AJAX princípu by sa mal server používať len ako zdroj a úložisko dát). Tento presun je spôsobený stále narastajúcimi požiadavkami na User Experience webových aplikácii[12] a faktom že web sa stal najrozšírenejšiu platformou. Prehliadače dokážu stále efektívnejšie interpretovať JavaScript a webové aplikácie stále viac preberajú funkčnosť natívnych aplikácii.

[8] Bližšie bude popísaný v kapitole 6

[9] V literatúre je možné nájsť aj pojem serverový framework

[10] Bližšie rozoberiem v kapitole 2.3.4

[11] Týka sa to hlavne aplikačných webových frameworkov bežiacich v cloude

[12] Webové aplikácie by sa mali chovať ako desktopové aplikácie

Presunom aplikačnej logiky zo servera na stranu klienta sa teda zmenila pôvodná úloha prehliadačov. Prehliadače pôvodne zobrazovali iba statický obsah HTML dokumentov. Dnes je úloha prehliadača komplexnejšia a prehliadač sa stará o aplikačnú logiku a vykonávanie zmien UI na základe interakcie s užívateľom.

Tieto zmeny sa uskutočňujú pomocou JavaScriptu, často s použitím nejakého UI frameworku (React, Angular, Vue.js). Tento prístup má za následok zvýšenie požiadavkov na výkon na strane klienta - prehliadača. [62]

Súčastný trend presunu aplikácii do prostredia webu ma nasledovné výhody a nevýhody:

+ Web je v súčastnosti najuniverzálnejšia a najrozšírenejšia platforma.
+ Nie je potrené si inštalovať žiaden runtime pre beh aplikácie (Ako v prípade .NET alebo Java aplikácii)
+ Kód aplikácie beží v sandboxe, čo je prostredie ktoré je odtienené od systému. Toto prostredie je určené pre spúšťanie nedôveryhodného kódu (Hosťovský operačný systém je chránený).
– Aplikačná vrstva musela byť napísaná v JavaScripte. [13]
– Strata niektorých platformovo špecifických vlastností

Presun desktopových aplikácii do prostredia webového prehliadača uľahčujú napríklad tzv. Single Page Aplikácie (Single Page App - SPA) a Progresívne Webové aplikácie (Progressive Web App - PWA).

Single Page App (SPA)

SPA je webová aplikácia ktorá pre prerendrovanie obsahu stránky robí bez toho aby server požiadala o nový HTML dokument. Ak webová aplikácia potrebuje nové data, aby mohla prerendrovať UI, požiada o to server pomocou AJAX. V prehliadači je AJAX engine, ktorý sa stará o užívateľskú interakciu a komunikáciu so servrom. AJAX engine pomocou DOM API[14] následne upravuje UI a data. Príklad SPA aplikácie je uvedený v sekcii MVC 2.1.4. AJAX engine SPA stránky musí obsahovať tzv. router. Router sa stará o zachytávanie požiadaviek na presmerovanie v SPA aplikácii. Toto zachytenie umožňuje vytvoriť stránku bez nutnosti požiadavky na server o HTML súbor. Router sa tiež stará o správne načítanie stránky po vložení URL do adresného riadku prehliadača.[89][100]

Progressive Web App (PWA)

Definícia PWA je na internete veľmi rôznorodá a občas nejednoznačná. Zjednodušene môžeme povedať, že PWA je webová aplikácia ktorá sa správa ako natívna aplikácia.

[13]Dlhú dobu bol JavaScript jediný natívne podporovaný jazyk

[14]Jedná sa o programové rozhranie k UI. DOM bude bližšie popísaný v kapitole 2.3.1

Každá PWA aplikácia však musí mať nasledovné vlastnosti:

- Musí bežať offline (musia sa načítať všetky URL webovej aplikácie)
  - Musí používať Service Workers (umožňuje načítanie zdrojov stránky z offline cache)
  - Musí používať TLS (prerekvizita pre Service Workers)
- Musí mať tzv. Web App Manifest[15].

Nasledujúce vlastnosti nie sú označené ako nutné, je však vhodné aby ich webová aplikácia mala.

- Responzívita - aplikácia by mala umožnovať navigáciu bez navigačných možností prehliadača (adresný riadok - URL bar).
- Rýchle načítanie aplikácie - Prvé načítanie, pred inštaláciou Service Workers by malo byť pod 10s. Po nainštalovaní Service Workers byť spustenie aplikácie malo byť pod 2s.
- Užívatelia aplikačne by mali byť informovaný že aplikácia pracuje offline.

Service Worker ktorý som spomínal vyššie je script ktorý beží na pozadí prehliadača. Tento skript sa pri prvom načítaní stránky nainštaluje do klientskeho prehliadača.

Service Worker sa následne správa ako proxy server medzi našou webovou aplikáciou a serverom. Umožňuje programovo pristupovať ku cache a IndexDB prehliadača a uložiť do nej potrebné súbory a dáta. Tieto súbory môžu byť následné použité pre beh aplikácie offline.

Správanie PWA môžeme zhrnúť nasledovne: Pri prvom načítaní stránky má užívateľ možnosť nainštalovať PWA aplikáciu. Po nainštalovaní aplikácie sa stiahne HTML dokument aplikácie[16] a uloží sa skript Service Workera. Po spustení PWA aplikácie komunikuje táto aplikácia výhradne skrz Service Workera. Service Worker pri inštalácii uloží do cache všetky potrebné súbory. Príklady PWA aplikácie - Twitter, Instagram, atď.[30][41][81][85][86][27]

Zhrnutie SPA a PWA

SPA prístup umožnil vytvárať webové aplikácie ktoré používajú server iba ako zdroj dát a biznis logiky. Tento prístup vznikol po predstavení AJAX a znamenal zmenu pohľadu na webové aplikácie.

PWA je súboroch technológii ktoré umožňujú webovej aplikácii bežať bez prítomnosti servera. Tieto aplikácie používajú server ako zdroj dát, na rozdiel od SPA však dokážu s užívateľom interagovať aj bez prítomnosti pripojenia na internet.

[15]Ten slúži prehliadaču na to, aby rozpoznal že sa jedná o PWA aplikáciu a obsahuje start_url čo je url k hlavnému HTML dokumentu

[16]Súbor ktorý sa nachádza v Web manifeste na adrese **start_url**

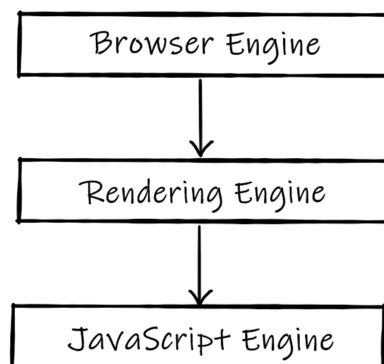
Vývoj webových aplikácii je veľmi komplexný a preto je nutná elementárna znalosť základných technológii. Tieto technológie sú HTML, CSS, DOM a Javascript. V podkapitolách 2.3.1, 2.3.2, 2.3.3 a 2.3.4 budú tieto technológie popísané. Programovací jazyk JavaScript budem popisovať podrobne pretože je dôležitým prerekurzorom vzniku WebAssembly a tiež popisuje určité špecifické vlastnosti kvôli ktorým vznikla WebAssembly.

Klientske technológie sú špecifické tým, že je pri nich kladený dôraz na kompatibilitu s klientskými prehliadačmi. Táto kompatibilita sa zabezpečuje dodržiavaním odsúhlasených štandardov organizácii W3C, WHATWG a ECMA. Pri nedodržaní týchto štandardov prehliadačom je pravdepodobne, že daný prehliadač stratí užívateľskú základňu a zanikle podobne ako Netscape Navigator a Internet Explorer.

V nasledujúcich kapitolách sa budem venovať základným stavebným prvkom, ktoré sú potrebné pre beh webovej aplikácie na strane klienta - v prehliadači.

2.3.1 Webový prehliadač

Pre spustenie webovej aplikáciu, potrebujeme webový prehliadač. Webový prehliadač je aplikácia ktorá umožňuje užívateľovi interagovať s webovou aplikáciou. Prehliadač sa zjednodušene skladá z nasledovných častí:



Obr. 2.4: *Zjednodušená schéma architektúry prehliadača*

Browser engine sa stará o dotazovanie a manipuláciu nad Rendering engine. Rendering engine sa stará o zobrazenie HTML dokumentu užívateľovi a obsahuje DOM, bližšie popísaný v kapitole 2.3.1. Rendering engine používa pre JavaScriptové a WebAssembly súbory JavaScript engine, ktorý obsahuje JIT kompilátor a interpreter JavaScriptu. Čo sa týka definície Rendering a Browser engine, v literatúre sa často mylne tieto dva pojmy zamieňajú. Browser engine môžeme zjednodušene považovať za interface cez aký pristupujeme k Rendering engine.

Prvé prehliadače obsahovali vlastné implementácie jednotlivých časti, dnes stále viac prehliadačov používa ako Browser engine open-source Chromium[17]. Pre príklad, Google Chrome, Opera, Vivaldi a Brave používajú Chromium ako svoj Browser Engine.[99][37][71] Chromium sa skladá z Rendering engine **Blink** a JavaScript engine **V8**.

Nasledujúca tabuľka obsahuje zoznam existujúcich Browser engine a ich Render a JavaScript engine-ov.

Tab. 2.1: *Prehľad Browser Engines*

Browser engine	Rendrovací engine	JavaScriptový engine
WebKit	WebCore	Nitro (JavaScriptCore)
Chromium	Blink	V8
Servo	Stylo	SpiderMonkey
EdgeHTML	EdgeHTML	ChakraCore
Gecko	WebCore	SpiderMonkey

Microsoft v decembri 2018 oznámil že majú v pláne nahradiť v prehliadači Microsoft Edge Browser engine za Chromium. pre Browser engine EdgeHTML to však neznamená zánik, stále sa používa v UWP aplikáciach (vrátane PWA aplikácií) ktoré sa nachádzajú v Microsoft Store. JavaScript engine ChakraCore je OSS a v súčastnosti sa často používa ako samostatný JavaScript engine. Tento JavaScript engine uviedol napríklad Time Travel debugging, ktoré umožnuje pri debuggovaní programu „krok späť“. Táto vlasnosť je zatiaľ dostupná iba pri použití JavaScript engine ChakraCore[18].[8][54][75]

Domain Object Model (DOM)

Dôležitou súčasťou každého prehliadača je Domain Object Model, ktorý umožňuje programový prístup k HTML dokumentu pomocou JavaScriptu. Tento model nepredpisuje použitie žiadneho konkrétneho programovacieho jazyka, v praxi sa však výhradne používa JavaScript.

Okolo roku 1995 počas vojny prehliadačov vznikla prvá špecifikácia, nazývaná DOM Level 0. Táto špecifikácia neumožňovala dynamicky dopĺňať pomocou DOM objektu dáta, umožnila však napríklad klientskú validáciu na **form** elemente. V roku 1997 bola konzorciom W3C vydaná nová špecifikácia DOM, s prívlastkom Dynamic HTML, ktorá je kolekciou technológii ktoré umožnili dynamické zmeny HTML dokumentu pomocou DOM API. Táto špecifikácia umožnila vznik technológii AJAX

[17] Chromium sa označuje browser a aj Browser engine

[18] JavaScript engine V8 to zatiaľ nepodporuje, `https://github.com/nodejs/diagnostics/issues/164`

(popisovanej v kapitole 2.1.3) a znamenala začiatok tzv. „dot com bubliny“ ktorá znamenala extrémne zvýšenie používania internetu firmami a tým aj zvyšovanie prostriedkov ktoré boli vynakladané na vývoj nových technológii. [57][46]

2.3.2 HyperText Markup Language (HTML)

Je to značkovací jazyk, tzv. markup, ktorý bol vytvorený pre popis štruktúry webovej stránky. Tento markup môže obsahovať určité elementy z ktorých sa skladá webová stránka. Každý element je reprezentavný tzv. tagom, ktorý ho popisuje. Webové prehliadače nezobrazujú HTML tagy, ale rendrujú ich obsah do stránky. Existuje 6 hlavných verzií HTML, kde každá nová verzia HTML pridáva nové tagy. Ukážka HTML5 kódu je na výpise kódu 2.1.

Tento jazyk je odvodený z jazyka SGML a vytvorili ho Tim Berns-Lee a Robert Cailliau na konci roku 1990. Popisoval prvých 18 elementov HTML. [9][96]

Tab. 2.2: Verzie HTML

Verzia	**Rok vydania W3C špecifikácie**
HTML	1991
HTML 2.0	1995
HTML 3.2	1997
HTML 4.01	1999
XHTML	2000
HTML5	2014

2.3.3 Cascading style sheets (CSS)

Je to jazyk ktorý bol vytvorený pre definíciu zobrazenia HTML dokumentov. Jeho hlavný účel je oddelenie grafickej reprezentácie dát a ich obsahu. Prvé špecifikácie HTML obsahovali elementy, ktoré určitým spôsobom formátovali obsah dokumentu. Vytvorenie týchto elementov sa považuje za chybu, pretože elementy majú hovoriť o štruktúre dokumentu a nie o zobrazení tejto štruktúry. Tento jazyk bol vytvorený Håkonom Wium Liesom a Bertom Bossom. Napriek tomu že v tej dobe existovalo viacero štýlovacích jazykov pre HTML dokumenty, bolo konzorciom W3C doporučené pre štýlovanie HTML dokumentov práve CSS vo verzii 1.

Aktuálna verzia na ktorej sa pracuje je CSS 3, ktorá bola kvôli svojej komplexnosti rozdelená na tzv. moduly. Tieto moduly sú postupne spracovávané a pridávané do špecifikácie. Niektoré zdroje uvádzajú chybne CSS4[5], ktorá ale neexistuje a jedná sa o modul CSS 3 - „CSS Background & Borders Level 4“. Ukážka CSS kódu je na výpise kódu 2.1 v elemente **style**.

Tab. 2.3: *Verzie CSS*

Verzia	Rok vydania W3C špecifikácie
CSS 1	1996
CSS 2.0	1998
CSS 2.1	2011
CSS 3	Rozpracovaná verzia k 6.12.2018 `https://www.w3.org/TR/2018/WD-css-align-3-20181206/`

```
1  <!DOCTYPE html><!-- HTML5 hlavička -->
2  <html>
3  <head>
4      <title>Canvas-Rotation</title>
5      <meta charset="UTF-8" />
6      <!-- style element umožňuje inline vloženie CSS do HTML-->
7      <style>
8          /* CSS komentár*/
9          #square {
10             border: 1px solid black;
11             transform: scale(10) rotate(3deg) translateX(0px);
12             -moz-transform: scale(10) rotate(3deg) translateX(0px);
13         }
14     </style>
15 </head>
16 <!-- HTML komentár-->
17 <body>
18     <canvas id="square" width="200" height="200" />
19 </body>
20 </html>
```

Výpis kódu 2.1: *Ukážka kódu HTML5 a CSS*

W3C vydáva tzv. snapshots tejto špecifikácie, ktoré väčšina majoritných prehliadačov implementuje. Návrhy je možné podávať prostredníctvom GitHub issue na adrese `https://github.com/w3c/csswg-drafts/issues`.[10]

2.3.4 Javascript

Web je najviac rozšírená aplikačná platforma na svete a z historických dôvodov bol jediný natívne podporovaný programovací jazyk tejto platformy práve JavaScript.

Javascript je vysokoúrovňový, multiparadigmatický, interpretovaný, dynamický a prototypovo založený programovací jazyk. Vznikol v roku 1995 ako skriptovací jazyk pre prehliadač Netscape Navigator. Po prvých úspešných verziách prehliadača

Obr. 2.5: *Oficiálne logo JavaScriptu.*

Netscape, chcela spoločnosť Netscape Communication vytvoriť viac dynamický web. Potrebovali skriptovací dynamický jazyk ktorý by sa dal vložiť priamo do HTML dokumentu a tým umožnil dizajnérom a grafikom vytvoriť interaktívnu stránku.

História a dôvody vzniku

V dobe, keď sa vedenie Netscape Communication rozhodlo vytvoriť tento nový skriptovací jazyk, najali pána Brendana Eicha. Pôvodne bolo jeho úlohou zaintegrovať programovací jazyk Scheme do prehliadača Netscape Navigator. Firma sa však nakoniec rozhodla po rokovaniach so Sun Microsystems[19] integrovať do prehliadača programovací jazyk Java.

Pána Brendana Eicha teda namiesto integrovania Scheme do prehliadača Netscape Navigator poverili vytvorením prototypu nového skriptovacieho jazyka. Takto vznikol jazyk pod kódovým označením Mocha. Prototyp Mocha bol vytvorený za 10 dní a požiadavky pri návrh prototypu boli nasledovné:

1. Musí to byť jazyk, ktorý je vložený v HTML
   - Musí byť interpretovaný
   - Musí byť rýchly na načítanie
   - Musí byť tolerantný voči chybám
2. Musí byť jednoduchý = dostupný pre neprogramátorov - „amatérov“
   - JavaScript bol primárne určený pre grafikov a dizajnérov
3. Syntax musí byť podobná programovaciemu jazyku Java

V decembri 1995 sa po uzatvorení dohody medzi Netscape Navigator a Sun Microsystems zmenil názov jazyka na dnes známy JavaScript. Tento názov mal reflektovať vtedajšiu nadvládu platformy Java v enterprise sektore. Zároveň vznikol článok, kde 28 „vedúcich“ spoločností (AT&T, Apple Computer, Inc., Macromedia, Oracle Corporation, Novell, Inc., atď.[20]) schválilo JavaScript ako doplnok k Jave pre jednoduchý

[19]Firma ktorá vytvorila programovací jazyk Java a vlastnila k nemu práva, túto firmu kúpil Oracle

[20]Microsoft nebol prizvaný pretože vyvíjal konkurenčný prehliadač Internet Explorer

online vývoj webových a intranetových aplikácii.

Brendan Eicha povedal, že JavaScript nebol navrhnutý pre vytváranie aplikačnej vrstvy. Úlohou JavaScriptu boli menšie úlohy pretože Java bola príliš komplikovaná a zložitá. JavaScript mal teda slúžiť iba ako skriptovací doplnok k Jave, čo by sa analogicky prirovnať vzťahu Visual Basic a C/C++ na platforme Windows.

Vývoj prototypu tohto jazyka veľmi ovplyvnil aj tlak, ktorý vytváralo vedenie firmy Nescape Navigator. Brendan Eich mal okrem vyššie uvedených požiadaviek v návrhu vlastností jazyka voľnú ruku. Jazyk dostal syntax podobnú Jave, ale má silné základy vo funkcionálnej Scheme a prototypo orientovanom Self.

Webové stránky začali rýchlo dostávať prvé interaktívne prvky, ako klientskú validáciu, tlačítka atď. Tieto interaktívne prvky fungovali iba v prehliadači Netscape Navigator, čím získal konkurenčnú výhodu.

Internet Explorer 3 naimplementoval podporu pre JavaScript v auguste 1996. Táto implementácia bola vytvorená pomocou tzv. „reverse engineeringu", pretože neexistovala žiadna oficiálna špecifikácia. Implementácia Microsoftu nedostala názov JavaScript ale JScript, kvôli možným súdnym sporom o slovo Java. Keďže samotný JavaScript sa stále vyvíjal, bolo správanie týchto dvoch implementácii veľmi nekonzistentné.

```
1  //javascript komentár
2  var canvas = document.createElement('canvas');
3  canvas.width = 200;
4  canvas.height = 200;
5  var image = new Image();
6  image.src = 'images/card.png';
7  image.width = 114;
8  image.height = 158;
9  image.onload = window.setInterval(function () {
10     rotation();
11 }, 1000 / 60);
```

Výpis kódu 2.2: *Ukážka Javascriptu*

Netscape sa obával veľkej fragmentácie implementácii JavaScriptu. Na internete sa začali objavovať stránky ktoré niesli logo Internet Exploreru alebo Netscape Navigátoru. Tieto logá hovorili o tom, pre ktorú implementáciu XScriptu je stránka optimalizovaná.

V novembri 1996 teda začal spolupracovať s Microsoftom a v júni 1997 spolu vytvorili prvý verziu špecifikácie. Táto špecifikácia vyšla v verzii 1 pod označením ECMA-262 pod dozorom medzinárodnej organizácie ECMA International so sídlom v Ženeve vo Švajčiarsku.

Kvôli obchodnej značke JavaScript sa táto špecifikácia nazvala podľa ECMA organizácie - ECMAScript. Organizácia W3C nikdy nevydala doporučenie pre používanie JavaScriptu na Webe. **Fakt ale je že JavaScript bol 20 rokov**[21] **jediný natívne podporovaný jazyk webovej platformy.** Tento fakt znamenal postupnú zmenu jeho pôvodného určenia z jazyka pre neprogramátorov na tzv. assembly language webovej platformy.

Pojmom assembly language myslím jazyk ktorý je najbližší platformovému strojovému kódu. Tak ako je špecifický assembler pre x86 alebo ARM architektúru, tak **bol**[22] špecifický JavaScript pre webovú platformu. JavaScript sa kvôli tomuto veľmi často sa používal ako tzv. compilation target, čo znamená že sa do neho kompilovali rôzne iné jazyky (Java - GWT, Python, C#, atď.).

JavaScript je od iných jazyk charakteristický aj tým že rôzne „implementácie“[23] JavaScriptu medzi sebou súťažili a dopĺňali sa, čím sa formovala aj ECMA špecifikácia JavaScriptu na ktorej tieto konkurenčné strany spolupracovali.

Vďaka vyššie spomínaným faktom vznikol pre JavaScript obrovská komunita a ekosystém. To že má JavaScript textovú podobu znamená aj to že sa všetky jeho zdrojové kódy dajú jednoducho čítať a upravovať.[4][77][73][101][80][12]

Špecifiká JavaScriptu

Počiatočné JavaScriptové engine boli iba interpreteri JavaScriptu do strojového kódu platformy na ktorej bežali (x86, ARM). Tento prístup bol ale pomerne neefektívny a tak vznikla myšlienka vytvoriť bytecode ktorý bude reprezentovať javascriptový kód. Tento bytecode je možné optimalizovať a Just in Time kompilátorom kompilovať do strojového kódu. Týmto prístupom vznikol aj pojem JavaScript engine, pretože už to nebol iba interpreter, ale Virtual machine s vlastným JIT kompilátorom (ten sa stará o optimalizáciu JS).

Tento prístup je principálne rovnaký pre tradičné VM ako JavaVM a CLR. Rozdiel je však v tom, že tradičné VM dostanú bytecode (CIL) a tento potom vykonávajú pomocou JIT kompilátora. JS engine musí spraviť 2 kroky, „skompilovať“ JS kód do bytecode a ten potom môžu používať tak ako tradičné VM.

Google prišiel v roku 2008 so svojim enginom V8 s prístupom, kde odstránil bytecode. V8 totiž miesto generovania a optimalizácie bytecode vytvárala priamo strojový kód. Toto vynechanie medzikroku zrýchlilo vykonávanie JavaScriptu 10-nasobne. Tento prístup je rýchlejší ako prvé dva, má ale jednu nevýhodu. Je plne závislí na tom, ako je napísaný JavaScripový kód.

[21] Od roku 1997 až do roku 2017

[22] WebAssembly prebrala tento prívlastok

[23] Týka sa to JavaScript enginov a rôznych API

Pre príklad uvediem prácu poľa čísel a optimalizácie V8 pre metódy `map`, `forEach`, `reduceRight`. Polia `array1` a `array2` z výpisu 2.3 sú oproti poľu `array3` z výpisu 2.4 podstatne optimalizovanejšie. Optimalizácie pre `array1` a `array2` sú ekvivalentné.[14]

Ak by sa takýto kód kompiloval AOT, optimalizácie by boli rovnaké. Presné fungovanie optimalizácii závisí aj od použitého JavaScriptového engine[24]. Niektoré JavaScripové engine používajú kvôli vyššie spominanému problému kombináciu strojového kódu a bytecode prístupu.

```
1 const array1 = [1,2,3]
2 const array2 = new Array(0)
3 array.push(1)
4 array.push(2)
5 array.push(3)
```

Výpis kódu 2.3: *Optimalizácia v JavaScriptovom engine V8 bude prebiehať podľa kritérií pre* ***PACKET_SMI_ELEMENTS***

```
1 const array3 = new Array(3)
2 array3[0] = 1
3 array3[0] = 2
4 array3[0] = 3
```

Výpis kódu 2.4: *Optimalizácia v JavaScriptovom engine V8 bude prebiehať podľa kritérií pre* ***HOLEY_ELEMENTS***

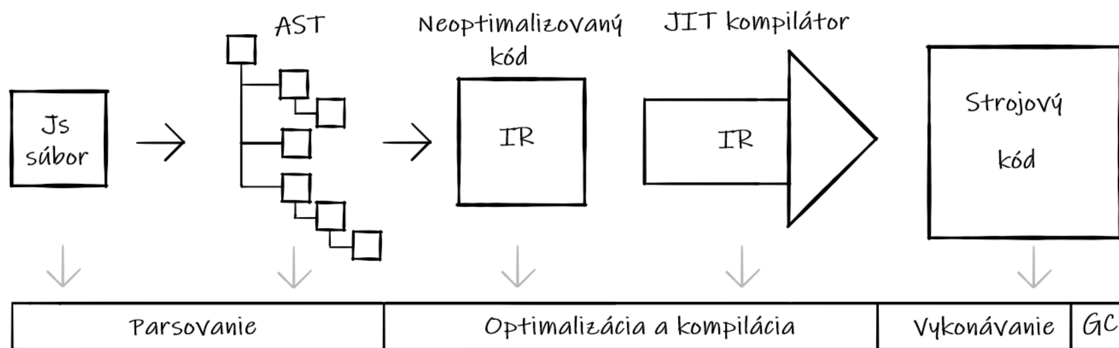
Problematika JavaScripových engine je veľmi komplexná téma a preto budem popisovať princíp enginu, na momentálne najrozšírenejšom engine - V8. Na obrázku 2.6 sa nachádza schéma JavaScripového enginu a vykonávania JavaScripového kódu (súboru). Pred spustením tohoto javascripového kódu musí byť dostupná celá textová reprezentácia kódu (súbor).

Na schéme 2.6 je vidieť koľko strávi JS engine na parsovaní, optimalizácii a kompilácii JS kódu do strojového kódu platformy. Samotné vykonávanie kódu je z tohoto časového úseku menej ako 1/4.

Zo schémy vyplýva, že **JavaScript nie je vhodný ako assembly language a ani ako compilation target** pretože JavaScripový engine trávi väčšinu času na optimalizáciách a parsovaní JS kódu. Problémy kompilácie rôznych jazykov do JavaScriptu budem bližšie popisovať k kapitole 3.1.

Pre príklad Engine V8 rieši vyššie spomínaný optimalizačný problém tak, že engine obsahuje 2 kompikátori. Jeden kompilátor je veľmi rýchly a produkuje neoptimálny strojový kód. Paralelne s týmto kompilátorom beží druhý kompilátor, ktorý pri opakovanom volaní funkcie optimalizuje strojový kód.

[24]Prehľad prehliadačov a ich JavaScript engine je v tabuľke 2.1



Obr. 2.6: *Principiálna schéma spúšťania JavaScripového kódu v JavaScriptovom engine spolu s časovým pomerom jednotlivých krokov*

Napriek tejto snahe, JavaScript stále nedosahuje výsledky ako natívny kód pri práci s grafikou, šifrovaním a prácu s audio-video. Komunita si uvedomila že potrebujú assembly language ktorý bude mať binárnu podobu a statický typový systém a jazyk musí byť kompilovaný AOT. Týmto by totiž zabezpečila rýchlosť ktorá by sa dala porovnať s rýchlosťou natívneho kódu.[97][37][17][18]

JavaScript Object Notation (JSON)

Je to jednoduchý textový formát, ktorý je určený na výmenu dát. Je ľahký na parsovanie a generovanie. Umožňuje stromový zápis dát z objektov. Do JSON dokážeme serializovať objekty a polia. Oproti XML ma výhodu v jednoduchosti a je veľmi používaný pri AJAX a REST API.

3 WebAssembly

„Wasm je binárny inštrukčný formát pre zásobníkový virtuálny počítač. Wasm je dizajnovaný ako portable compilation target pre vysokoúrovňové jazyky ako C, C++, Rust a umožňuje vývoj pre webové klientske a serverové aplikácie.„

Oficiálna stránka	`https://webassembly.org`
Oficiálny repozitár na GitHub-e	`https://github.com/webassembly`
W3C pracovná skupina	`https://www.w3.org/wasm/`

Táto technológia reprezentuje dôležitý vývojový stupeň evolúcie vývoja webových aplikácii. Umožňuje spúšťať skompilovaný a optimalizovaný kód v prostredí webového prehliadača bez nutnosti inštalácie externého pluginu. WebAssembly je portable, veľkostne a load-time efektívny formát určený pre beh v sandboxe.

WebAssembly umožňuje zmeniť prístup k vývoju webových aplikácii, ktorý doteraz znamenal pre webovú aplikáciu používať JavaScript[1].

Ako som spomínal v kapitole 2.3.4, pre spustenie JavaScriptového kódu v prehliadači je nutné najskôr celý súbor stiahnuť, následne vytvoriť syntaktický strom (AST), priradiť typy premenným a určitý overhead má aj inicializácia JIT (optimalizácia a deoptimalizácia, pamäť, atď.).

Nevýhody JavaScriptu ako assembly jazyka je teda jeho textová forma a nedeterministická optimalizácia kódu. JavaScript je navrhnutý pre ľudí a WASM je navrhnutá pre X engine[2], aby ho dokázal čo najefektívnejšie spustiť a optimalizovať kód ktorý X engine spúšťa.

Obr. 3.1: *Oficiálne logo WebAssembly, chýbajúca vrchná časť má znázorňovať, že sa jedná o „doplnok“.*

WebAssembly sa dá v jednoduchosti zhrnúť ako nízkoúrovňový jazyk podobný assembleru, založený na stack machine, ktorý je určený pre spustenie v JavaScrip-

[1]Týka sa to aj rôznych supersetov JavaScriptu (Typescript)

[2]Zámerne som sa vyhol JavaScripovému enginu pretože Wasm má aj vlastné WASM engine

tovom engine. Tento jazyk je silne typový a podporuje iba číselné typy (i32, i64, f32 a f64).

V špecifikácii existujú 2 formy tohto kódu, binárna (WASM) a textová (WAT). Textová forma je reprezetácia binárnej formy a je určená pre debbuging a rôzne nástroje na analýzu kódu. Binárna forma je určená pre produkčné použitie a spúšťanie v JavaScriptovom engine[3].

WebAssembly môžeme označiť aj ako compilation target pre web, bola od začiatku vyvíjaná ako doplnok k JavaScript-u, ktorý je s ním plne kompatibilný a beží v JavaScriptovom engine. S JavaScriptom dokáže spolupracovať na základe *WASM JavaScript API*. WebAssembly je vyvýjaná ako webový štandard pod vedením W3C (`https://www.w3.org/wasm/`), s podporou majoritných prehliadačov (Mozilla, Chrome, Safari,Edge).[61]

Táto technológia bola prvýkrát predstavená v roku 2015. Hlavnou úlohou WASM je podporiť 3D rendering, editáciu videa, hry, kompresiu, atď. vo webovom prehliadači. [103][98][102]

3.1 História a dôvody vzniku

Snaha o klientskú interakciu v prehliadači nielen pomocu JavaScriptu, je zjavná z technológií Java Applet, Silverlight, alebo Adobe Flash. Tieto technológie neboli úspešné kvôli tomu, že fungovali ako pluginy do prehliadačov a neboli ich súčasťou.

Webové aplikácie majú stále väčšiu JavaScriptovú codebase čo znamená nárast veľkosti súborov. Tento nárast veľkosti má za následok to, že medián interaktivity stránok pre mobilné zariadenia sa pohybuje okolo 9,1s a medián načítania každého JavaScriptového súboru je 0,6s.[4] Tieto fakty nútia webovú komunitu hľadať iné spôsoby ako optimalizovať technológie, ktoré sa používajú v prehliadačoch.

Pluginy JavaApplet a Silverlight mali výhodu vlastného runtime (JVM alebo CLR) v hosťovskom OS prehliadača, a tak dokázali vykonávať kód takmer natívnou rýchlosťou.

Prerekurzormi vzniku WebAssembly by sa dali nazvať pokusy skompilovať rôzne jazyky do JavasScriptu, čo malo svoje výhody a nevýhody medzi ktoré patrí:

+ Statická a silná typovosť (niektorých) kompilovaných jazykov a ich tooling pri vývoji.
+ Compile-time kontrola chýb.
+ Statická analýza kódu (pri dynamických jazykoch je to veľmi náročné).
+ JavaScriptové engine-y dokážu pomerne efektívne spúšťať aj veľké skompilované JavaScriptové codebase.

[3]Existujú stand-alone WebAssembly runtime ako napríklad `https://wasmer.io/`

[4]Údaje sú aktuálne k 19.04.2019 (zdroj: `https://httparchive.org/reports/loading-speed`)

– Pri debuggovaní aplikácie a hľadaní chyby bolo nutné, aby vývojár poznal JavaScript a jeho tooling.
– Výsledok kompilácie je často neefektívny (súvisí to aj s vlastnosťami JavaScriptu).

V roku 2006 Google uvoľnil GWT (Google Web Toolkit[5]), čo je v skratke súbor nástrojov pre kompiláciu aplikácie napísanej v Jave do JavaScript-u. Pomocou tohto nástroja sú napísané Google Maps, Gmail alebo populárna platforma Blogger[6]. Táto technológia sa stále vyvíja.

Existuje veľmi veľa jazykov, ktoré je možné skompilovať do JavaScriptu, stále je ale treba rátať s **nevýhodami** tejto kompilácie:

– Ignorácia sémantických rozdielov skompilovaného jazyka a JavaScriptu.
– Generovaný JavaScript nie je tak optimálny, ako kód, z ktorého je vygenerovaný (čo sa môže prejaviť na výkone, alebo pri debuggovaní).

Myšlienka kompilácie rôznych jazykov do JavaScriptu je teda plne funkčné a často praktické riešenie, nerieši však problém výkonu JavaScriptu pre špecializované použitia, ako sú napríklad video a grafika.

JavaScriptové engine ako V8 (Google), Spidermonkey (Mozilla) alebo Chakra (Microsoft)[7] dokážu pomocou optimalizácie JavaScripotvý kód skompilovať a rýchlejšie „interpretovať“, čím sa zvyšuje efektívna rýchlosť Javascriptu.

Jedna z optimalizácií je napríklad pridanie typu premennej (ukážka kódu 3.1)[8], čo spôsobí, že JavaScriptový engine nemusí zisťovať typ „za behu“, ale zistí ho vo fáze načítania skriptu, pri vytváraní AST, čím sa zefektívni jeho činnosť.[13]

```
1 function compiledCalculation() {
2     var x = f()|0;  // x je 32-bit int hodnota
3     var y = g()|0;  // y je 32-bit int hodnota
4     return (x+y)|0; // 32-bit sčítanie, nevykonáva sa žiadna kontrola typu alebo
      ↪ pretečenia
5 }
```

Výpis kódu 3.1: *Ukážka JavaScriptového kódu s explicitne definovanými typmi premenných*

Ako sa z webu stáva platforma pre budovanie distribuovaných aplikácii, je potrebné budovať komplexné aplikácie, ktoré musia byť rýchle (už pri prvom načítaní

[5] http://www.gwtproject.org/
[6] https://www.blogger.com/
[7] Microsoft Edge bude používať JavaScripový engine V8 a renderovací engine Blink[7], kvôli tomu, že prechádza na Chromium
[8] JavaScript síce nemá priamo **typ int 32** ale je možné ho použiť pomocou bitwise operátora | a väčšina JavaScripových engine interne tento typ používa

je požadovaná plná funkcionalita stránky).

Google a Mozilla, tvorcovia prehliadačov Chrome a Firefox, sa rozhodli vyvinúť technológie, ktoré by riešili tieto problémy. Spoločnosti sa rozhodli pre odlišné prístupy k riešeniu vyššie uvedeného problému a vznikli dve technológie **Native Client** a **asm.js**.

3.1.1 Native Client, Portable Native Client

Netscape vytvoril v 90. rokoch NPAPI (Native Pepper API), ktoré umožňovalo komunikovať pluginom prehliadača so systémom „priamo". Pomocou NPAPI mohol plugin volať akékoľvek systémové volanie, čo je obrovská bezpečnostná diera, a preto sa táto technológia zavrhla. Tento prístup používali aj technológie JavaApplet a Silverlight.

Mozilla vytvorila PPAPI (Portable Pepper API), v rámci projektu Mortar, ktoré pôvodne malo priniesť PDFium [9] a Flash pluginy do prehliadača Firefox. Rozdiel oproti NPAPI bol ten, že cieľ kompilácie nebola architektúra hosťovského zariadenia prehliadača, ale LLVM IR kód [10]. Pre predstavu, LLVM IR kód nie je strojový kód, ale určitý medzikód, ktorý je následne pomocou kompilátora v prehliadači skompilovaný na strojový kód. Mozilla, ktorá spolupracovala na projekte s Google ale projekt zavrhla.

Google v tomto projekte pokračoval a ďalej ho rozvíjal a vytvoril Native Client (NaCl). NaCl bola možnosť ako spustiť natívny kód vo vnútri prehliadača, čiže v sandboxe. Pre to aby sa kód nemusel pre každú architektúru kompilovať zvlášť, Google použil Portable Native Client (PNaCl). Tieto technológie sa skladájú z 2 elementov.

1. Toolchain[11] ktorá kompiluje C/C++ kód do tzv. NaCl/PNaCl modulov.
2. Runtime komponenty, ktoré sú integrované v prehliadači a umožňujú spustenie NaCl/PNaCl modulov (pomocou PPAPI a NPAPI).

Po vytvorení modulov **nexe** (NaCl) a **pexe** (PNaCl) bolo možné tieto moduly spustiť v prehliadači. Interakcia modulu s prehliadačom bola možná vďaka vyššie spomínanému Pepper API (NPAPI a PPAPI). Štandard tohoto API bol však nekompletný a chýbala dokumentácia, takže jediný prehliadač, ktorý podporoval túto technológiu bol Google Chrome (a Chromium).

Výhoda tohoto prístupu spočíva v tom, že z C/C++ sme pomocou LLVM toolchainu dokázali vytvoriť veľmi výkonný natívny kód (pretože používal inštrukčnú sadu danej architektúry - AMD64, x32, ARM).

[9] PDF reader pre prehliadač

[10] LLVM a LLVM IR bližšie popíšem v kapitole LLVM 3.1.4

[11] Použitý LLVM s CLang kompilátorom

V Googli sa začala debata či pokračovať v NaCl, pretože sa to veľmi podobalo ére pluginov[12] a po debatách medzi V8 tímom a NaCl tímom sa Google rozhodol v máji 2017 oficiálne ukončiť tento projekt a vývojárov presunúť do novovzniknutého WebAssembly tímu.[29][39][16][50][72]

3.1.2 asm.js

Mozilla sa rozhodla vydať po ukončení spolupráce na PPAPI cestou optimalizácie JavaScriptu a vznikol asm.js, čo je **subset** JavaScriptu, v ktorom sa používajú iba **číselné dátové typy**.

Obmedzenie na číselné dátové typy má za následok to, že JavaScriptový engine nemusí pri JIT kompilácii JavaScriptového kódu zisťovať typy premenných, pretože tieto typy sú známe už z AST. Táto optimalizácia sa vykonáva pri kompilácii JavaScriptového/C/C++ kódu do asm.js, t.j. Ahead of Time (AOT).

Asm.js je teda **typový** textový subset JavaScriptu, ktorý je optimalizovaný tak, aby JavaScript engine po vytvorení Abstract Syntax Tree nemusel zisťovať dátový typ premenených.

Formát asm.js sa môže použiť ako compilation target, pretože umožňuje optimalizácie, aké samotný JavaScriptový kód neumožňuje. Myšlienka pridať silnú typovosť do JavaScriptového kódu je v ostrom kontraste s vlastnosťou JavaScriptu, ktorou je jeho dynamická typovosť. Rýchlosť asm.js kódu bola podľa benchmarkov iba o 50% nižšia ako natívna rýchlosť[13].[93]

```
1 function f(i) {
2     return i + 1;
3 }
```

Výpis kódu 3.2: *Kód pred optimalizáciou pomocou Emscripten*

```
1 "use asm";
2 function f(i) {
3     i = i|0;
4     return (i + 1)|0;
5 }
```

Výpis kódu 3.3: *Kód po skompilovaní do asm.js formátu pomcou nástroja Emscripten*

Výpisu kódu 3.3 je vo formáte asm.js. Direktíva na prvom riadku „`use asm`" hovorí JavaScriptovému engine, že nasledujúci kód je v asm.js formáte. Ak engine

[12] Bolo treba do prehliadača pridať NaCl/PaCl plugin, a ten nebol štandardizovaný

[13] Rýchlosť, ktorá by bola ekvivalentná rýchlosti ak by bol kód napísaný v strojovom kóde danej platformy

podporuje asm.js, tak pri vytváraní AST priradí premenným typy, ktoré sú definované pomocou bitwise „|" operátoru a typu „0", ktorý za ním následuje (0 znamená 32 bitový integer). Ak engine nepodporoval asm.js, tak sa tento kód vykonával ako štandardný JavaScript.

Subset asm.js je teda optimalizovaný subset JavaScriptu, kde sú polia a premenné implicitne typové a používajú sa v nej iba numerické dátové typy. Výkon asm.js sa mierne približuje výkonu natívnemu kódu. asm.js má síce výhody skompilovaného jazyka, ale stále sa jedná o JavaScript, čiže nie je možné plne využiť všetky optimalizácie, ktoré ponúka LLVM kompilátor.[106][53][105][29][97][39]

Vlastnosti asm.js možno zhrnúť do nasledujúcich bodov:

- asm.js kód predchádza „spomaľovaniu" kódu pomocou implicitného typu premenných. Toto umožňuje AOT (Ahead of Time) kompilácia kódu do JavaScriptu (asm.js) a následne zrýchlenie JIT kompilácie JavaScripového engine.
- JavaScriptový engine má garanciu, že typ danej premennej sa nebude v runtime meniť, takže môže vygenerovať efektívny strojový kód.

3.1.3 Zhrnutie Native Client a asm.js

Native Client priniesol výkon natívneho kódu do prehliadaču a asm.js zase portabilitu kódu, ktorý je možné spustiť bez pluginu v každom prehliadači. Fúziou týchto myšlienok vznikla WebAssembly ktorá je architektonicky agnostická ako asm.js, ale zároveň umožňuje kódu výkon podobný Native Client. Native Client a aj asm.js sú popri Flash Player, Java Applet a Silverlight míľniky, ktoré boli prerekurzormi technológie WebAssembly. Obrovskú zásluhu na rozvoji WebAssembly má Mozilla, ktorej prehliadač je lídrom v implementácii štandardu W3C WebAssembly.[29]

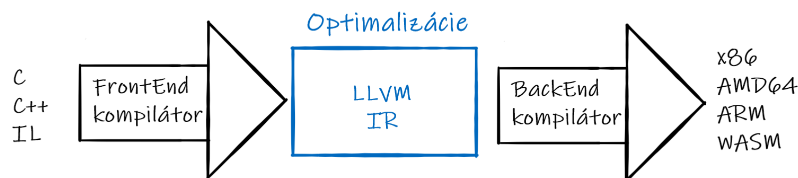
Nečakaný výkon asm.js mal za následok, že sa vytvorila skupina z Mozilla, Microsoft, Google a Apple a vytvorili pre tento subset JavaScriptu binárnu reprezentáciu. Táto binárna reprezentácia sa stala základom pre štandard WebAssembly.

3.1.4 LLVM

Skratka LLVM pravdepodobne vznikla z **L**ow **L**evel **V**irtual **M**achine. LLVM je toolchain kompilátorov, ktoré dokážu deterministicky a efektívne optimalizovať kód. Tento toolchain vznikol ako modulárny toolchain pre vytváranie kompilátorov pre rôzne jazyky a compilation targets (x86, AMD64 a WASM).

Na schéme 3.2 je princíp fungovania LLVM toolchainu. Na vstupe je kód, ktorý je napísaný v rôznych vysokoúrovňových jazykoch (C++, C a IL[14]). Tento kód je pomocou frontend kompilátoru skompilovaný do Intermedia Represenatation, ktorý

[14]Intermedia Language bude popísaný v kapitole 4.1



Obr. 3.2: *Princíp fungovania LLVM toolchainu*

je následne optimalizovaný. Táto optimalizovaná forma je následne skompilovaná pomocou backend kompilátora do compilation target.

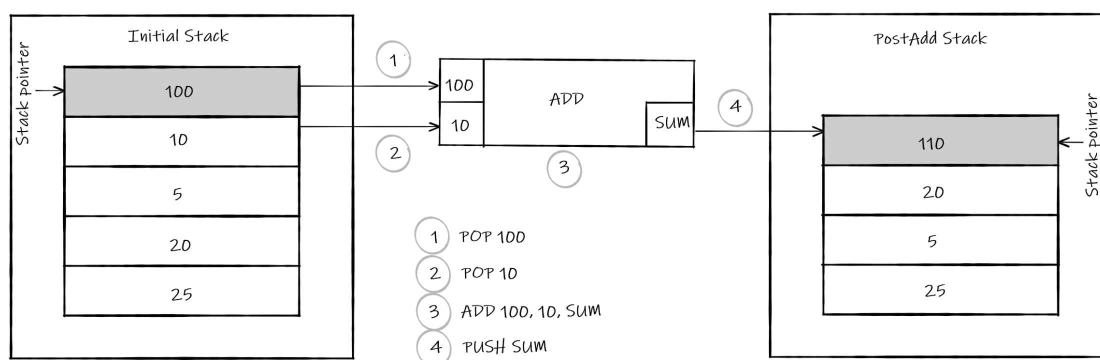
Toto rozdelenie a modulárnosť LLVM má obrovské výhody:

+ Pre pridanie podpory nového vysokoúrovňového jazyka je nutné vytvoriť iba frontend kompilátor.
+ Pre pridanie podpory nového compilation target je nutné vytvoriť iba nový backend kompilátor.
+ Pre podporu rôznych optimalizácií je možné pracovať iba s IR kódom, čím sa tieto optimalizácie unifikujú skrz rôzne jazyky.

Jeden z compilation targets je aj WebAssembly čo umožňuje rôznym jazykom, ktoré majú frontend kompilátor používať WebAssembly ako compilation target a bežať v prehliadači.[51]

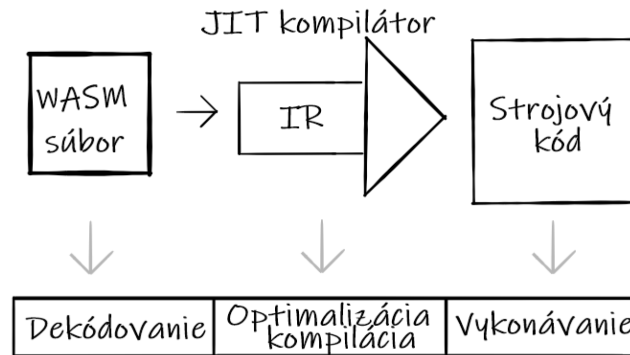
3.2 Princíp fungovania WebAssembly

WebAssembly funguje na princípe virtuálneho zásobníkového počítača, pretože to umožňuje kompaktnejší kód.



Obr. 3.3: *Zjednodušená schéma zásobníkového počítača - Stack machine*

O WebAssembly môžeme povedať, že jej inštrukčný set architektúry (ISA) je RISC (Reduced Instruction Set Computer).



Obr. 3.4: *Principiálna schéma spúšťania WebAssembly v JavaScriptovom engine spolu s časovým pomerom jednotlivých krokov*

Na schéme 3.4 sa nachádza principiálna schéma spúšťania WebAssembly v X[15] engine. Engine pri načítaní súboru tento súbor iba dekóduje a vytvorí z neho IR kód. **Tento kód nie je LLVM IR kód**. Tento IR slúži JIT kompilátoru na optimalizácie a následnú kompiláciu na inštrukčnú sadu danej platformy.

WebAssembly je síce binárny formát, ale špecifikácia hovorí aj o textovej WAT (WebAssembly Text) reprezentácii, ktorá slúži pre tooling a ľudí. V praxi sa nepredpokladá, že by sa písala WebAssembly ručne pomocou WAT formátu. WAT zápis používa S-expression, ako je vidieť na zápise 3.4. Tento kód je reprezentáciou C kódu z výpisu kódu 3.5.

```
1  (module
2    (type $t0 (func))
3    (type $t1 (func (result i32)))
4    (func $__wasm_call_ctors (type $t0))
5    (func $main (export "main") (type $t1) (result i32)
6      i32.const 42)
7    (table $T0 1 1 anyfunc)
8    (memory $memory (export "memory") 2)
9    (global $g0 (mut i32) (i32.const 66560))
10   (global $__heap_base (export "__heap_base") i32 (i32.const 66560))
11   (global $__data_end (export "__data_end") i32 (i32.const 1024))
12 )
```

Výpis kódu 3.4: *WAT zápis binárneho WASM súboru vygenerováneho z kódu 3.5*

Tieto výpisy kódu sú spracované pomocou textového editora VsCode[16] a NPM „balíčku“ @wasm/studio-utils[17].[59][66][90][70]

[15] Môže sa jednať o JavaScripový alebo WebAssembly engine
[16] https://code.visualstudio.com
[17] https://www.npmjs.com/package/@wasm/studio-utils

```c
#define WASM_EXPORT __attribute__((visibility("default")))
WASM_EXPORT
int main() {
  return 42;
}
```

Výpis kódu 3.5: *Program v jazyku C pred kompiláciou*

Použitie v prehliadači

WebAssembly monentálne nie je možné v prehliadači spustiť napriamo. Jediná možnosť je načítanie WebAssembly modulu cez JavaScript a následné používanie tohoto modulu pomocou JavaScript interoop. Výpis kódu 3.6 ukazuje načítanie WebAssembly modulu zo súboru **main.wasm**, ktorého WAT reprezentácia je na výpise 3.4.

```javascript
let wasmPath = '../out/main.wasm';
fetch(wasmPath).then(response =>
    response.arrayBuffer()
).then(bytes => WebAssembly.instantiate(bytes)).then(results => {
    instance = results.instance;
    document.getElementById("container").textContent = instance.exports.main();
}).catch(console.error);
```

Výpis kódu 3.6: *Zavolanie funkcie main() pomocou JavaScript interoop*

Ak by prehliadač nepodporoval WebAssembly, je možné použiť fallback na asm.js. Tento fallback je ale veľmi nepraktický, pretože výsledný súbor vo formáte asm.js môže mať desiatky MB, čo je pre prostredie webu veľmi nepraktické. Keďže je WebAssembly štandard W3C a podporuje ho už takmer 85% prehliadačov[18], nie je tento spôsob fallbacku praktický.

Ak by sme chceli pri kompilácii wasm súborov vytvoriť aj asm.js súbory, je to možné pomocou nástroja Emscripten, ktorý používa na kompiláciu vyššie spomínaný LLVM toolchain.

3.3 Porovnanie WebAssembly a JavaScriptu

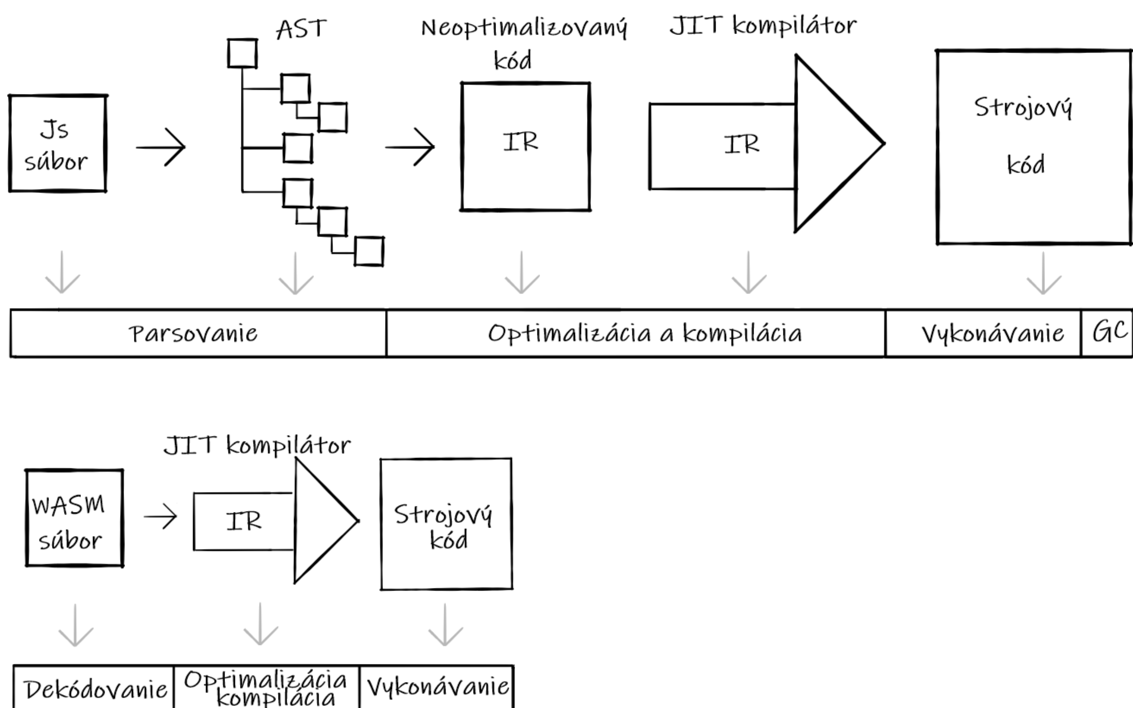
Nasledujúci zoznam obsahuje výhody, ktoré má WebAssembly oproti JavaScriptu.

+ Rýchlejšie načítanie binárneho súboru, ako súboru s JavaScriptom (aj keď je použitá komprimačná technológia gzip).

[18]zdroj: `https://caniuse.com/#feat=wasm`

+ Dekódovanie binárneho WASM súboru do AST je rýchlejšie ako parsovanie JavaScriptových súborov.
+ Kompilácia a optimalizácia kódu je rýchlejšia, pretože WASM je bližšie k strojovému kódu. Časť optimalizácií je tiež možné urobiť už pri kompilácii do WASM.
+ Reoptimalizácia - na strane kleinta nie je potrebná, pretože WebAssembly má „vbudované“ typy a ostatné informácie potrebné pre beh. JavaScript engine teda nemusí optimalizovať WASM kód a tým pádom nie je potrebná réžia na optimalizáciu, ako v prípade JavaScriptu.
+ Vykonávanie kódu zaberá menej času (aj kvôli vyššie spomenutým výhodám).

Na schéme 3.5 je vidieť časový pomer medzi vykonávaním JavaScriptového a WebAssembly kódu. Zo schémy je pomerovo vidieť o koľko je vykonávanie WebAssembly rýchlejšie. Dekódovanie súboru je oveľa rýchlejší proces ako parsovanie, pretože engine nemusí zisťovať typy premenných a vytvárať komplexný AST. Jednoduchší AST a implicitné typy premenných znamenajú rýchlejšiu JIT optimalizáciu a vykonávanie kódu.[20][21]



Obr. 3.5: *Porovnanie vykonávania JavaScriptového súboru a WASM súboru s časovým pomerom jednotlivých krokov*

3.4 Štandardy W3C a podpora prehliadačov

Aktuálny stav W3C štandardov je taký, že zatiaľ existujú iba tzv. „drafts“, čiže návrhy, ktoré ešte neboli schválené a vydané W3C konzorciom ako doporučenia (stav k 22.4. 2019).

Tab. 3.1: *Štandardy W3C pre WebAssembly*

WebAssembly Core Specification	`https://www.w3.org/TR/wasm-core-1/`
WebAssembly JavaScript Interface	`https://www.w3.org/TR/wasm-web-api-1/`
WebAssembly Web API	`https://www.w3.org/TR/wasm-js-api-1/`

Aktuálne sa pracuje na 3 špecifikáciách. Prvá špecifikácia s názvom *WebAssembly Core Specification*, hovorí o základných vlastnostiach WASM ako sú formát kódu, jeho efektívne vykonávanie a jeho kompaktnosť. Druhý dokument s názvom *WebAssembly JavaScript Interface*, hovorí o spolupráci medzi WASM a JavaScriptom. Posledný dokument s názvom *WebAssembly Web API* popisuje širšiu integráciu WebAssembly s webovou platformou. Tabuľka 3.1 obsahuje odkazy pre jednotlivé dokumenty.

Prvý z prehliadačov, ktorý začal podporovať WebAssembly, bol Firefox vo verzii 52. Keďže všetky štandady W3C sú stále rozpracované, nasledovné informácie nemusia byť aktuálne a vzťahujú sa na obdobie písania tejto diplomovej práce.[19]

Čo sa týka plnej podpory špecifikácie, tak plnú podporu vyššie spomínaných dokumentov majú zatiaľ iba Firefox (v62) a Chrome (v69)[20].

Prehliadače **Opera, Safari a Edge** momentálne **neimplementujú** globálny objekt `WebAssembly.Global`, ktorý slúži na dynamické linkovanie viacerých modulov skrz import a export.[60][61][103]

3.5 Aplikácie ktoré používajú WebAssembly

Existuje pomerne veľké množstvo projektov, kde sa využíva WebAssembly, jedná sa hlavne o výkonovo náročnejšie projekty:

- Unity (3D engine) - `https://unity3d.com/learn/tutorials/s/tanks-tutorial`
- Oryol (3D framework) - `https://floooh.github.io/oryol/`
- PyBullet (Real-Time Physics Simulation engine) - `https://github.com/kripken/ammo.js`
- D3-force (Graph Simulation library) - `https://github.com/ColinEberhardt/d3-wasm-force`

[19] Aktuálne ku dňu 15.5. 2019

[20] Spolu prehliadačmi postavenými na Chromium

- ONNX.js (Open Neural Network Exchange)- `https://github.com/Microsoft/onnxjs`

Ako je vidieť zo zoznamu, jedná sa zatiaľ o veľmi špecifické použitie WebAssembly, do budúcna sa však ráta s pridaním Garbage Collectoru a rôznych „vyššie úrovňových“ jazykov.

3.6 Budúcnosť a rozpracovaná funkcionalita

Ako som spomínal v kapitolách vyššie, WebAssembly je stále vo vývoji a momentálne je to tzv. MVP (Minimum Valuable Product). V začiatkoch WebAssembly stojí nepochybne Emscripten, ktorý umožnil skompilovať C++ do JavaScriptu, čím sa začal vývoj asm.js, ktoré vyústilo až do vyvinutia WebAssembly.

MVP WebAssembly dosiahol nasledové ciele:

- Kompiláciu do neutrálneho jazyka[21], niečo ako Assembler, ale s tým, že tento jazyk nie je určený pre fyzickú architektúru.
- Rýchle vykonávanie tohto jazyka JavaScriptovým engineom
- Kompaktnosť kódu, čo je dosiahnuté jeho binárnou formou.
- Lineárna pamäť - stack machine, čo je zabezpečenie voči tomu, aby WebAssembly kód mohol pristupovať iba k pamäti, ktorú mu určí JavaScriptový engine (pomocou TypedArray, čo je pole, ktoré obsahuje bajty pamäte).

Ďalší cieľ je pridať podporu pre tzv. ťažké desktopové aplikácie, ako napríklad Photoshop alebo AutoCAD. Pre podporu takýchto aplikácii je potrebné pridať funkcionalitu, ktorá je popísaná v zoznamoch *Využitie vlastností moderného hardware* a *Vylepšenie loadtime WebAssembly v prehliadači*.

WebAssembly je určená aj pre štandardný vývoj webových aplikáci. Požiadavky na fukcionalitu sú popísané v zozname *Vylepšenia pre štandardný vývoj webových aplikácii*.

Pre lepšiu predstavu o aktuálnom stave funkcionality som pridal hodnotenie 0 až 4.[22] Význam hodnotenia je nasledovný:

0. - funkcionalita nemá draft a prebieha o nej diskusia.
1. - pripravuje sa draft funkcionality.
2. - funkcionalita má draft a prehliadače ju začínajú implementovať.
3. - funkcionalita má draft a niektoré prehliadače ju experimentálne implemenujú.
4. - funkcionalita je schválená štandardom W3C.

---

[21] V anglickej literatúre sa to označuje pojmom „language-agnostic“

[22] Hodnotenie je založené na zázneme prednáške WASM Post-MVP Future od Lil Clark z konferencie CovalenceConf 2019 zverejneného dňa 22.01. 2019

Využitie vlastností moderného hardware

- Threading - pre prácu s modernými typmi procesorov je potrebné pridať podporu pre vlákna, čím sa otvára cesta k viac-jadrovým procesorom. **(3)**
- SIMD - je to ďalšia možnosť paralelného spracovania dát, pretože 1 inštrukciou dokážeme spracovať viacero dát.**(2)**
- Podpora AMD64 architektúry - s architektúrou x86(32bit) je k dispozícii „iba" 4GB adresovateľnej pamäte. 64-bitová adresa pamäte umožnuje používať 16 exabajtov pamäte.**(0)**

Vylepšenie loadtime WebAssembly v prehliadači

- Streaming compilation - WebAssembly (JavaScript engine) dokáže spúšťať WASM súbor postupne, tak ako prichádzajú pakety - tým sa WebAssembly ešte zrýchli.**(3)**
- Tiered compilation - odstupňovaná kompilácia, ktorá umožňuje pomocou viacnásobnej kompilácie dooptimalizoavať výsledný kód. Táto technika sa často používa pri jazykoch, ktoré pre svoj beh používajú runtime (.NET, Java).**(3)**
- Implicitná HTTP cache - z cache si prehliadač vyberie správny WASM súbor, z ktorého je už vytvorený strojový kód (o ktorý sa postaral JavaScript engine).**(3)**

Vylepšenia pre štandardný vývoj webových aplikácii

- Rýchle volania medzi JavaScriptom a Wasm. **(3)**
- Jednoduchá a rýchla výmena dát medzi JavaScripotom a Wasm. **(2)**
- Integrácia ES modulov (používanie import a export direktív). **(3)**
- Integrácia toolchain (balíčkovací systém, webpack, parcel, wasmpack)**(3)**
- Spätná kompatibilita **(3)**

Vylepšenia pre vysokoúrovňové jazyky

- Automatická správa pamäti - Garbage Collector (GC). **(3)**
- Exception Handling **(1)**
- Tail call - optimalizácia pre call stack volania**(1)**

S vylepšeniami popísanými vo vyšších zoznamoch, bude možné prepísať hlavné časti frameworkov ako Angular, Ember, React alebo Vue.js do WebAssembly. Tento prepis bude mať za následok zrýchlenie behu týchto frameworkov, pretože vo WebAssembly bude možné paralelizovať výpočty, ktoré vykonávajú tieto frameworky (výpočty pre VirtualDom a pod.). Pre React by to napríklad znamenalo prepis dom

diffing algoritmu, čo je algoritmus, ktorý porovnáva starý, reálny DOM a nový aktualizovaný, virtuálny DOM.

WebAsembly môžeme ale zhodnotiť ako štandard budúcnosti webového vývoja, ktorý sa stále rozvíja a má podporu komunity, tvorcov prehliadačov a tiež konzorcia W3C.[23][19][31][22]

4 .NET - Common Language Infrastructure

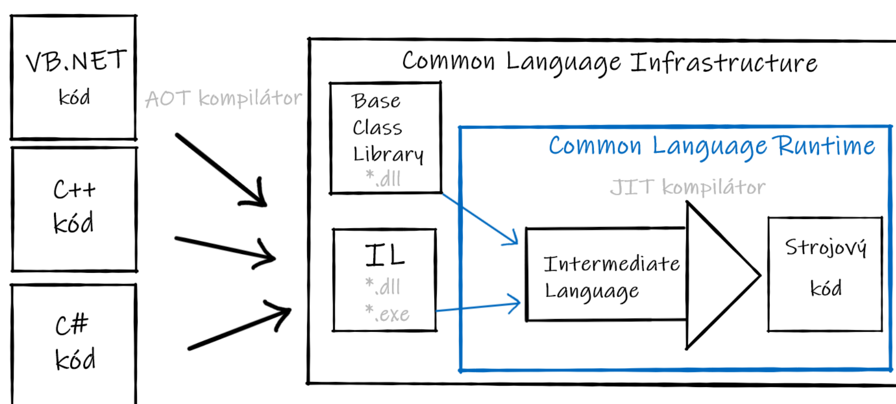
V 90. rokoch vzniklo v Microsofte veľa technológii z ktorých sa rozhodol Microsoft rozhodol v roku 2000 vytvoriť nový platformu pre vývoj aplikácii. Tie technológie mali pôvodne používať platformu Java, ale kvôli súdnym sporom sa rozhodol Microsoft vytvoriť vlastnú platformu a nazval ju .NET. Pre túto platformu vznikla špecifikácia, ktorá dostala názov Common Language Infrastructure (CLI). Nasledujúce kapitoly budú popisovať základné informácie o platforme .NET.

4.1 Common Language Infrastructure

Špecifikácia platformy CLI je systémovo agnostická a nepredpisuje použitie operačného systému ani použitie programovacieho jazyka.

Špecifikácia CLI sa skladá z nasledovných častí:

- **Intermediate Language (IL)** - tzv. bytecode ktorý používa CLR a pomocou JIT kompilátora ho optimalizuje a kompiluje na strojový kód
- **Common Language Runtime (CLR)** - Virtual machine pre IL kód (obsahuje JIT kompilátor)
- **Base Class Libraries (BCL)** - základná sada knižníc ktorá obsahuje všetky dôležité dátové typy (mscorlib) a metódy.



Obr. 4.1: *Principiálna schéma fungovania .NET platformy*

Technický štandard CLI Microsoft poskytol štandardizačným organizáciám ISO a ECMA, ktoré ho vydali pod ECMA-335[1] a ISO/IEC 23271[2].[63]

[1]Aktuálna verzia dostupná `https://www.ecma-international.org/publications/standards/Ecma-335.htm`

[2]Aktuálna verzia dostupná `https://www.iso.org/standard/36769.html`

4.2 .NET Framework

Prvú implementáciu CLI vytvoril Microsoft a dostala názov .NET Framework. Táto implementácia funguje iba pre proprietárny OS Windows, za čo bol Microsoft ostro kritizovaný. Pre .NET Framework vzniklo obrovské množstvo toolingu, ktorý uľahčuje vývoj aplikácii.

Pre príklad projekt Roslyn, čo je OSS kompilátor pre C# a VB.NET do IL kódu. Tento kompilátor je napísaný v C# -e čo umožňuje pokročilý refaktoring a analýzu kódu.

Microsoft z pôvodne proprietárneho prístupu k vývoju .NET Frameworku začal od roku 2008 postupne prechádzať do open-source filozofie. Z tejto iniciatívy vznikla nezisková organizácia .NET Foundation, ktorá zastrešuje všetky open-source projekty, ktoré súvisia s platformou .NET. Microsoft postupne uvoľňuje technológie ako WPF, Winforms a WinUI ako OSS. **.NET Framework je určený pre desktopové aplikácie OS Windows a pre serverové aplikácie ASP.NET, Windows Service a WebServices.** Pre túto platformu vznikol jazyk C# ktorý sa vyvíja v tandeme s .NET a je pre ňu najpreferovanejší.[63]

4.3 .NET Core

.NET Core je open-source (MIT licencia) implementácia CLI štandardu. Vznikol z iniciatívy Microsoftu, pretože proprietárny .NET Framework začal mať problémy s udržateľnosťou a implementáciou nových funkcii. Vývoj .NET Core je riadený komunitnou organizáciou .NET Foundation.

Oficiálna stránka	`https://dotnet.microsoft.com/`
Oficiálny repozitár na GitHub-e	`https://github.com/dotnet/coreclr`
Licencia	MIT

.NET Core sa skladá z CoreCLR (CLR), CoreFX (BCL). **Vývoj .NET Core bol od začiatku orientovaný na serverovú časť (cloud) a od verzie 3.0 sa rozširuje aj na desktopy.** Táto implementácia je charakteristická tým že má malý footprint v OS a je výkonejšia (V porovnaní s .NET Frameworkom).[63]

4.4 Mono

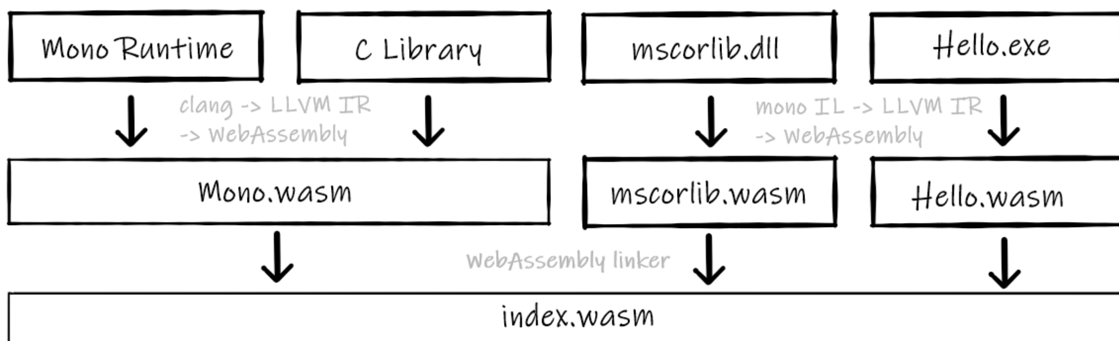
Mono je prvá open-source implementácia CLI infraštruktúry, ktorá umožňovania multiplatformový vývoj (Windows, Linux, MacOS). Táto platforma bola špecifická

aj svojim AOT kompilátorom ktorý dokáže C# kód skompilovať na špecifickú architektúru - compilation target (ARM, x86, LLVM IR, atď.). Ako som spomínal v sekcii 3.1.4, LLVM toolchain obsahuje backend kompilátor ktorý umožnuje skompilovať LLVM IR do Wasm formátu. Pri AOT kompilácii je dôležitý tzv. stripping, kedy IL linker dokáže z IL kódu odstrániť všetky nepotrebné funkcie, čo umožňuje vytvoriť kompaktný súbor vhodný pre web.

Oficiálna stránka	`https://www.mono-project.com/`
Oficiálny repozitár na GitHub-e	`https://github.com/mono/mono`
Licencia	MIT a 3-Clause BSD

V mojej diplomovej práci budem používať **Mono runtime**, ktorý je skompilovaný pomocou emscripten toolchainu[3] na strane servera do WASM súboru[4].[68][63]

Mono uviedlo v januári 2018 prototyp podpory AOT kompilácie. Táto kompilácia je určená pre produkčné prostredie, pretože umožňuje optimálnejší a rýchlejší beh IL kódu na klientovi, v prehliadači.



Obr. 4.2: *Principiálna schéma AOT kompilácie*

Schéma 4.2 graficky znázorňuje kompiláciu Mono runtime (C++) do WASM formátu, spolu s AOT kompiláciou IL (exe, dll) do wasm. Táto kompilácia bude bližšie popísaná v kapitole 5.1.1.

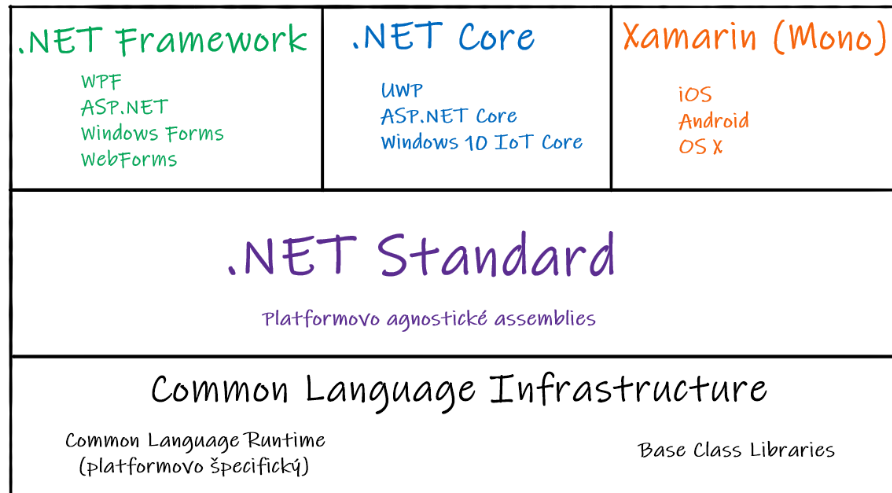
4.5 .NET Standard

Viacero implementácii CLI znamenalo roztrieštenosť aplikačných knižníc a kompilácie pre jednotlivé compilation targets na ktorých by mala bežať naša knižnica (.NETCoreApp, Mono, net462, i386-apple-darwin13.0.0, atď.). Spoločný prvok je síce IL, ale špecifické compilation targets umožňujú v kóde používať platformovo

[3]Tento toolchain používa LLVM

[4]Samotný runtime je napísaný v jazyku C++

špecifické API. Kvôli týmto problémom bolo nutné vytvoriť set spoločného API ktoré je pre všetky compilation targets rovnaké.



Obr. 4.3: *Grafické znázornenie .NET Standardu*

.NET Standard môžeme označiť ako sadu rozhraní ktoré implementujú všetky implementácie CLI. Ak napíšeme knižnicu ktorej pri kompiláci do IL určíme compilation target na `netstandard2.0` tak máme istotu, že kód ktorý napíšeme je možné používať zo všetkých implementácii CLI ktoré tento štandard implementujú.[63][64]

Tento štandard je dôležitý aj z hľadiska mojej implementácie, pretože umožňuje zdielať rovnaký kód medzi 2 verziami CLI runtimov - ASP.NET Core (CoreCLR - server) a Mono (Mono runtime - klient).

4.6 .NET a WebAssembly

Microsoft oznámil na konferencii Build zjednotenie CLI runtimov Mono, .NET Core a .NET Frameworku. Táto „nová“ implementácia CLI bude umožňovať ako jeden z compilation target aj WebAssembly.[28]

5 Blazor

Pre .NET vývojárov existuje mnoho možností pre fullstack[1] vývoj webových aplikácii. Niektoré technológie ako Silverlight alebo Web Forms už nie sú oficiálne podporované Microsoftom a preto si začala komunita vyvíjať rôzne alternatívy.

Microsoft po technológii WebForms nevyvíjal žiaden frontendový framework. .NET komunita preto prišla s projektmi ako Uno[2], DotVVM[3] a Cshtml5[4]. Microsoft v roku 2018 vydal OSS experimentálny fullstack framework Blazor. Dňa 18. apríla 2019 bola vydaná preview verzia s oficiálnou podporou zo strany Microsoft.[84]

Oficiálna stránka	`https://blazor.net/`
Oficiálny repozitár na GitHub-e	`https://github.com/aspnet/Blazor`

Výhody Blazor technológie súvisia hlavne v použitej .NET platforme a dajú sa zhrnúť nasledovne:

+ Tooling - .NET má veľmi bohaté nástroje, čo umožňuje pohodlný vývoj.
+ .NET ekosystém je stabilný a výkonný (rýchlosť a škálovateľnosť).
+ .NET-u je navrhnutý ako jazykovo neutrálny, čím umožňuje jazykovú diverzitu pre vývojárov.
+ Stabilita a konzistencia .NET platformy - platforma sa stále vyvíja ako open-source a spravuje ho nezisková komunitná organizácia .NET Foundation. Platforma je špecifická aj svojou spätnou kompatibilitou.

5.1 Špecifikácia Blazor

Blazor je SPA[5] framework postavený na ASP.NET Core a Mono. Pre UI používa template engine Razor, ktorá umožňuje kombinovať HTML a C# syntax. Blazor má byť alternatívou k JavaScriptovým SPA frameworkom ako React, Angular alebo Vue.js.

Technológia umožňuje 2 spôsoby prístupu k SPA aplikácii. Prvý je tzv. serverový pomocou WebSocketu[6] a druhý je tzv. klientský. V mojej práci sa budem zameriavať iba na klientský prístup, vzhľadom k tomu, že serverový prístup nie je pre moju prácu podstatný.

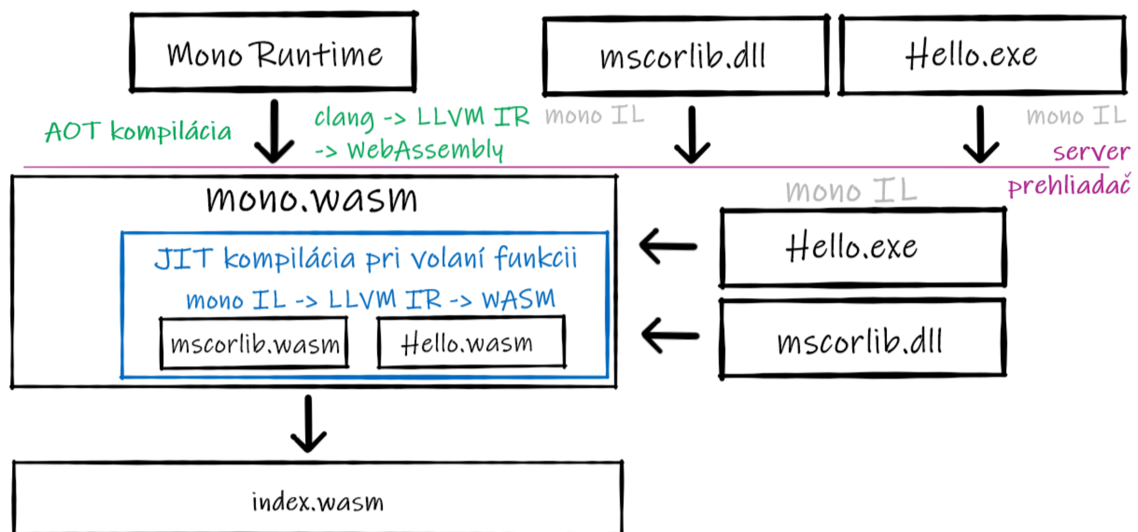
[1] Pojem hovorí o pojení frontend (klientských) a backend (servererových) technológii
[2] `https://platform.uno/`
[3] `https://www.dotvvm.com/`
[4] `http://www.cshtml5.com/`
[5] Single Page Application je popisovaný v 2.3
[6] Presnejšie Blazor používa SignalR

5.1.1 Blazor - klientsky prístup

Pre beh .NET knižníc v prostredí webového prehliadača sa používa odľahčená verzia Mono runtime. Ak by sme chceli zdielať rovnaký kód pre serverovú časť aplikácie a klientskú časť, ktorá beží v prehliadači, musíme pre kód použiť compilation target .NET Standard[7].

Blazor umožňuje 2 prístupy ako spustiť kód v prehliadači. Jeden som pre lepšie rozlíšenie nazval JIT prístupom a druhý AOT prístupom. JIT prístup umožňuje spustiť IL kód [8] v prehliadači priamo pomocou Mono runtime, ktorý obsahuje JIT kompilátor. JIT kompilátor obsahuje LLVM kompilátor a spolu cez postupnosť IL -> LLVM IR -> Wasm skompilujú kód do WebAssembly. Tento kód je následne cez JavaScriptový engine kompilovaný do strojového kódu architektúry (x86, ARM).

Princíp fungovania je graficky zobrazený na schéme 5.1. Fialovou čiarou sú oddelené vrstvy prehliadača a servera. Kompilácia, ktorá je nutná pred spustením aplikácie, je vyznačená zelenou farbou. Kompilácia, ktorá prebieha v prehliadači je vyznačená modrou farbou a v prípade JIT prístupu funguje kompilácia nasledovne: pri volaní funkcie runtime nájde správny IL kód a ten zoptimalizuje a následne preloží do WASM formátu (pomocou LLVM - sú tu 2 stupne optimalizácie JIT a LLVM). Tento prístup je preferovaný pre vývoj aplikácie, pretože umožňuje rýchlejší build.

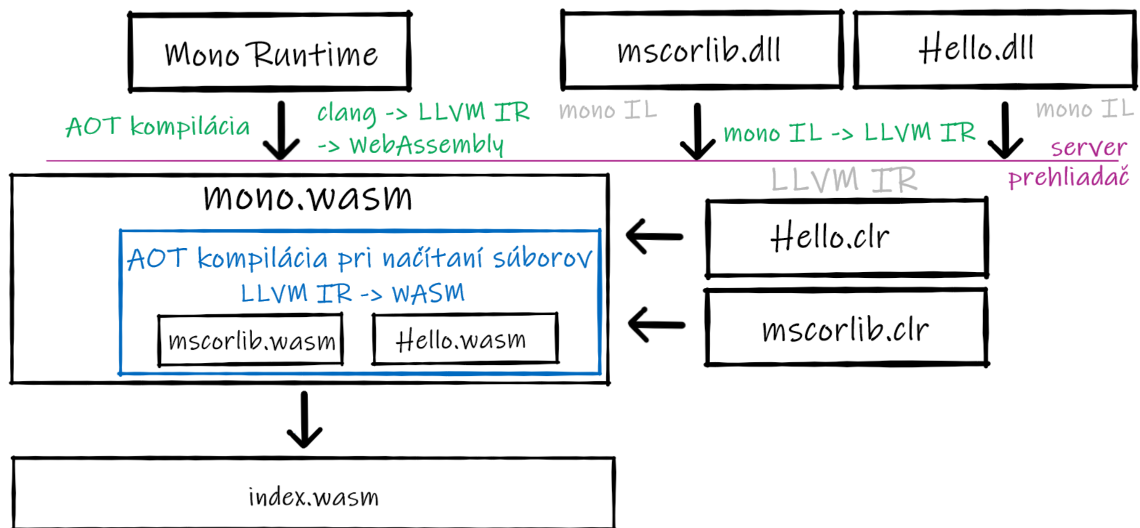


Obr. 5.1: *Principiálna schéma kompilácie kódu - JIT prístup.*

Druhý, AOT prístup umožňuje efektívnejšie vykonávanie, pretože optimalizácie vykonávané JIT kompilátorom sú vykonané pred spustením aplikácie. Tomuto spô-

[7]Na serveri máme ASP.NET Core 4.3 a na klientovi Mono runtime 4.4
[8].dll/.exe prípona s magic value MZ v hlavičke súboru

sobu zostalo označenie experiment. Tento prístup má nevýhodu v tom, že kompilácia trvá dlhšiu dobu a do prehliadača sa neposiela dll knižnica ale binárny .clr súbor. Tento súbor obsahuje LLVM IR kód. V mono.wasm súbore sa vďaka JIT optimalizácii mohol „vyhodiť“ JIT optimalizátor a zostal v ňom iba LLVM kompilátor. LLVM kompilátor zoberie LLVM IR, zoptimalizuje a skompiluje ho do wasm formátu. Tento prístup má **stále experimentálne označenie** a používam ho vo svojej diplomovej práci.



Obr. 5.2: *Principiálna schéma kompilácie kódu - AOT prístup*

Schéma 5.2 graficky, podobne ako predchádzajú schéma oddeľuje serverovú a klientsku časť fialovou čiarou. Kompilácie ktoré sú potrebné na serveri sú označené zelenou farbou a kompilácia v prehliadači (mono.wasm module) je zobrazená modrou farbou. Na serveri sa robí pomocou IL Linker **stripping**, čo je technika pri ktorej sa z kódu odstránia metódy a vlastnosti ktoré sa nepoužívajú. Následná kompilácia Mono IL do LLVM IR vytvorí optimalizácie pre JIT Mono runtime a výsledný IR kód je vložený do .clr súboru. Tento súbor je následne v prehliadači načítaný do mono.wasm runtime, ktorý LLVM IR kód skompiluje do wasm formátu a vykoná na ňom ďalšie optimalizácie. Vytvorený index.wasm je potom spustený ako wasm modul v JavaScriptovom engine prehliadača, ktorý ho kompiluje do strojového kódu. JavaScriptový engine vytvára tiež optimalizácie, ktoré sú ale špecifické pre engine[9].[88][26][52][87]

[9]Zoznam JavaScripových engine je v tabuľke 2.1

5.2 Aktuálny stav Blazoru

Projekt Blazor bol uvedený vo februári 2018 a od jeho vzniku bolo uvedených 7 minoritných verzíi (0.1.x až 0.7.0). Ako je vidieť z commit-ov v github repozitári `https://github.com/aspnet/Blazor`, tento projekt je veľmi aktívny a vývojári implementujú každú novinku z WebAssembly alebo Mono-a.

Projekt je od 18.4. 2019 vo verzi preview ale niektoré jeho časti (AOT kompilácia) sú stále pod označím experimental. Blazor je súčasť OSS ASP.NET Core repozitára ktorý spavuje .NET Foundation[10], čím sa zabezpečuje jeho ďalší vývoj a podpora.

[10] Organizácia má technickú riadiacu skupinu zloženú z Microsoftu, RedHatu, JetBrainsu, Unity, Samsungu a Googlu. Po komunitných voľbách v januári 2019 bol počet riaditeľov rozšírený z 3 na 7 a Microsoft má obsadenú iba jednu pozíciu[55]

6 DotVVM Framework

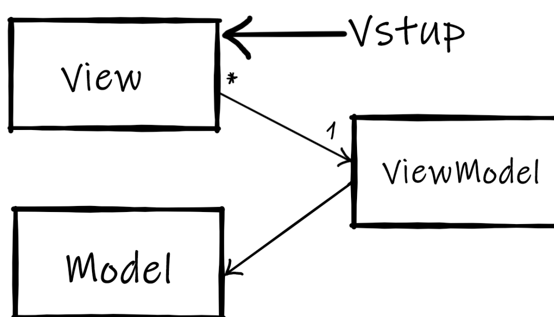
DotVVM je OSS aplikačný framework postavený nad ASP.NET Core, ASP.NET a knockout.js. Framework je kombinácia backend a frontend frameworku. Tak ako Blazor jedná sa o fullstack framework. Výhoda frameworku DotVVM spočíva v jeho jednoduchosti a jednoduchom prístupe k tvorbe UI.

DotVVM je špecifické tým, že používa návrhový vzor Model View ViewModel a AJAX. Framework uľahčuje písanie tzv. line of business aplikácií[1]. V nasledujúcej kapitole bude vysvetlená architektúra MVVM a jeho implementácia s frameworkom DotVVM, spolu so základnými konceptmi DotVVM.

6.1 MVVM

Architektonický návrhový vzor MVVM vytvoril architekt John Gossman pre technológiu WPF a Silverlight.

Tento vzor vychádza z MVC a jeho obrovská výhoda je v tom, že sa v ňom ľahko udržuje stav. Nositeľom stavu je totiž ViewModel, a ten je plne nezávislí na View, čím sa zabezpečuje jeho jednoduchá testovateľnosť. Úlohou View je zobrazovanie dát poskytnutých ViewModelom. View sa pri interakcii s užívateľom správa ako proxy, ktorá spracuje užívateľské vstupy a pre zmenu stavu prevolá príslušné metódy ViewModelu.



Obr. 6.1: *Principiálna schéma MVVM architektonického vzoru*

- **View** - je vstupným bodom do systému a stará sa o užívateľskú interakciu. Jeho úlohou je zobrazovať dáta ktoré poskytuje ViewModel. Užívateľskú interakciu preposiela na ViewModel pomocou **commands**.
- **Model** - v tomto vzore je Model braný ako dáta a doménová logika[2]. Túto časť je možné chápať ako zdroj dát (databáza, web API) ktoré sú poskytnuté.

[1] Napríklad interné informačné systémy, administratívne časti webových aplikácií, atď.
[2] Pravidlá podľa ktorých sa dáta vytvárajú, čítajú, upravujú a mažú

- **ViewModel** - riadiaca časť celého vzoru ktorá **udržuje stav**. ViewModel prijíma **commands** od View pomocou ktorých mení svoj stav (tu sa správa View ako proxy). Pomocou techniky data binding sa následne synchronizujú data medzi View a ViewModelom.

Výhody tohto vzoru sú teda jednoduché udržovanie stavu a jednoduchá testovateľnosť. Technika data binding umožňuje jednoduché zdieľanie hodnôt medzi View a ViewModel.

6.2 MVVM a DotVVM

Framework DotVVM emuluje architektúru MVVM medzi prehliadačom a serverom. Táto emulácia umožňuje v bezstavovom prostredí webu zdieľať rovnaký stav na základe ViewModelu. Na serverovej časti sa stará DotVVM o generovanie HTML dokumentov a inicializáciu ViewModelu hodnotami.

Na klientskej časti sa stará DotVVM o interakciu s užívateľom a o komunikáciu so serverom v prípade potreby dát alebo potreby zavolania **commands** pomocou **PostBack**-ov, čo sú AJAX volania na server s JSON formátom.

Zdieľanie rovnakého stavu medzi serverom a klientom umožňuje ViewModel.

V nasledujúcej časti budem podrobne popisovať ako funguje DotVVM na serveri a na klientovi - v prehliadači.

6.2.1 Základné koncepty

DotVVM aplikácia na serveri sa skladá z 2 hlavných častí. Prvá časť je View, ktorý je napísaný v DotVVM markupe, čo je derivácia jazyka HTML s vlastnými elementami a anotáciami. Vlastné elementy umožňujú používať komponenty ktoré obsahujú serverovú a klientskú logiku. Serverová logika sa stará tiež o vytvorenie správneho HTML kódu pre tieto komponenty. Klientská logika sa stará o vytvorenie správnych knockout.js bindingov, ktoré umožňujú previazanie ViewModel objektu a HTML pomocou DOM API.

Druhá časť je ViewModel ktorý je nositeľom logiky a stavu aplikácie. ViewModel je na serverovej časti napísaný ako trieda v jazyku C# . Pre DotVVM ViewModel platí že musí byť serializovateľný. Objekt ViewModelu je totiž prenášaný medzi serverom a klientom v JSON formáte[3].

Obmedzenie tohto prístupu pomocou serializácie je také, že sa interný stav objektu **neserializuje**. Pre DotVVM ViewModel teda platí že **sa zdieľa iba vonkajší (public) stav objektu**.

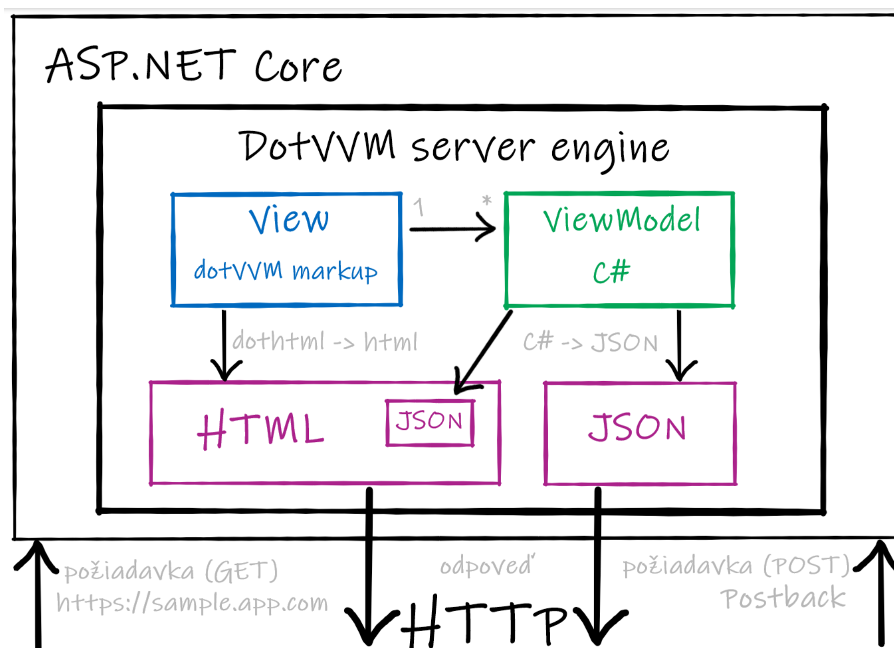
[3]Pre prácu s JSON formátom v C# sa využíva nuget balíček Newsoft.JSON

ViewModel obsahuje okrem stavu spravidla aj sadu funkcií, ktoré predvolávajú Model a tým menia stav aplikácie (ViewModel-u). Tieto funkcie sa volajú z klienta pomocou PostBackov. Pre lepšie pochopenie som rozdelil výklad fungovania DotVVM do 2 kapitol, kde budem najskôr vysvetľovať úlohu DotVVM na serveri 6.2.2 a následne úlohu DotVVM na klientovi - v prehliadači 6.2.3

6.2.2 Serverová časť

DotVVM engine na serveri pracuje v 2 režimoch[4]. Obidva prístupy sú znázornené na schéme 6.2.

Prvý z režimov je spracovanie klientskej požiadavky HTTP GET o HTML dokument[5] (ľavá strana obrázku). DotVVM engine na vyberie správny View na základe cesty „`https://sample.app.com`“ požiadavku. View má na seba naviazaný ViewModel ktorý môže obsahovať ďalšie ViewModel-e a vytvárať stromovú štruktúru ViewModel-ov s jedným koreňom. DotVVM engine serializuje ViewModel (strom) do JSON formátu a skompiluje View do HTML dokumentu. Do HTML dokumentu DotVVM engine vloží tiež serializovaný ViewModel, čím sa zabezpečuje uchovanie stavu ViewModelu zo servera.



Obr. 6.2: *Principiálna schéma fungovania DotVVM na serveri*

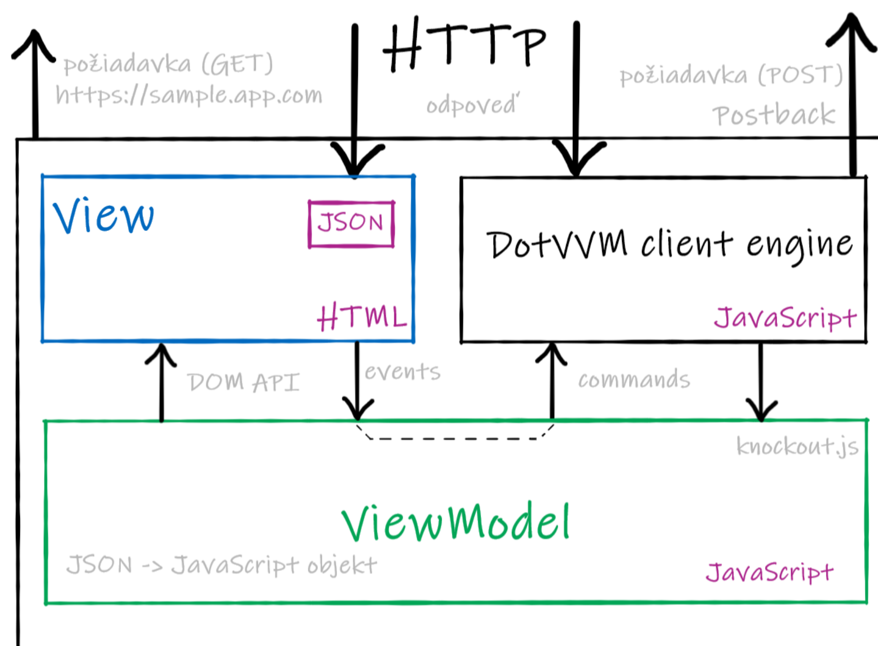
Druhý režim je spracovanie PostBack-u (pravá strana schémy). Ak v prehliadači urobí užívateľ akciu pre ktorú je aplikačný kód napísaný vo ViewModeli, klientský

[4]Framework umožňuje ešte aj iné prístupy, tie však v rámci zjednodušenia nebudem spomínať
[5]Prvá požiadavka užívateľa pri použití webovej aplikácie napísanej v DotVVM

DotVVM engine pošle na server PostBack (HTTP POST). PostBack obsahuje serializovaný ViewModel. DotVVM engine na serveri deserializuje tento objekt a vytvorí instanciu serverového ViewModelu (C# trieda). V tejto triede sa zavolá potrebná funkcia (ktorú užívateľ inicializoval). Ak sa zmení ViewModel, tak serverový engine DotVVM vytvorí rozdiel prijatého a aktuálneho (zmeneného) ViewModelu. Tento rozdiel je následne serializovaný do JSON a poslaný naspäť na klienta v HTTP odpovedi.

6.2.3 Klientská časť (prehliadač)

Po vyslaní prvej HTTP GET požiadavky z prehliadača, dostane prehliadač HTTP dokument so serializovaným ViewModelom. HTML dokument obsahuje odkazy na JavaScripové súbory DotVVM engine. Po načítaní skripov sa inicializuje DotVVM engine, ktorý zo serializovaného ViewModelu vytvorí pomocou knižnice knockout.js klientský ViewModel. View sa naviaže na stav ViewModelu pomocu DOM API. Ak užívateľ zmení na stránke nejakú hodnotu, je táto zmena pomocou knockout.js a DOM API prerendrovaná pre celú webovú aplikáciu. Ako som spomínal v predchádzajúcej kapitole 6.2.2, **všetky funkcie ViewModelu sú na strane servera** v C# triede[6].



Obr. 6.3: *Principiálna schéma fungovania DotVVM v prehliadači*

[6]Existujú ešte nasledovné možnosti: staticCommands, kompilácia C# do JavaScript

Funkcie na serveri sa z klienta prevolávajú pomocou PostBackov čo je volanie metódy POST. Na schéme 6.3 je zobrazené volanie funkcie ViewModelu. Interakcia začína ze View => event => ViewModel => Command => DotVVM client engine. DotVVM engine následne vyvolá PostBack na server, ktorý mu vráti rozdiel ViewModelov. Tento rozdiel je následne reflektovaný do ViewModelu. Knižnica knockout.js pri zmene ViewModelu zmení dokument pomocou DOM API. Rendering engine prehliadača následne prerendruje HTML dokument.

Úlohou mojej práce je vytvoriť spôsob ako minimalizovať počet PostBackov zo strany klienta na server, čiže volaní funkcií ViewModelu.

Časť II

Praktická časť

7 Prerekvizity

Nasledujúca kapitola obsahuje zoznam prerekvizít ktoré sú potrebné pre spustenie praktickej časti mojej diplomovej práce. Návod obsahuje postup pre inštaláciu Windows Subsystem for Linux, Mono, .NET Core SDK a Emscripten SDK, Node.js[1], libuvc a ninjabuild.

Pre inštaláciu častí na linux budem používať linuxový balíčkovací systém APT (Advanced Packaging Tool `https://help.ubuntu.com/community/AptGet/Howto`). Pre inštaláciu častí na Windows budem používať balíčkovací systém Chocolatey (`https://chocolatey.org/`).

7.1 Aktivácia WSL

Pred inštaláciou WSL je potrebné v OS Windows povoliť Windows rozšírenie pre Linux[2]. **Verzia build OS Windows musí byť väčšia ako 16215.** Verziu OS je možné zistiť nasledovný príkazom z PowerShell terminálu.

```
PS> systeminfo | Select-String "^OS Name","^OS Version"
```

Výpis kódu 7.1: *Zistenie verzie buildu OS Windows*

Ak je verzia vyššia ako 16215 je možné aktivovať WSL nasledujúcim príkazom:

```
PS> Enable-WindowsOptionalFeature -Online -FeatureName
↪   Microsoft-Windows-Subsystem-Linux
```

Výpis kódu 7.2: *Aktivácia WSL v OS Windows*

Po úspešnej aktivácii WSL je potrebné nainštalovať linuxovú distribúciu Ubuntu 18.04 LTS[3].

```
PS> Invoke-WebRequest -Uri https://aka.ms/wsl-ubuntu-1804 -OutFile .\ubuntu.zip
↪   -UseBasicParsing
PS> Expand-Archive .\ubuntu.zip .\ubuntu\
PS> .\ubuntu\ubuntu1804.exe
```

Výpis kódu 7.3: *Stiahnutie a inštalácia Ubuntu 18.04*

Inštalácia trvá približne 5 minút a po úspešnom nainštalovaní distra je nutné vytvoriť užívateľa.

[1] JavaScripový runtime pre spúštanie JavaScriptu mimo prehliadača `https://nodejs.org/en/`

[2] Podrobný návod je dostupný na adrese `https://docs.microsoft.com/en-us/windows/wsl/install-win10`

[3] Alternatívne je možné stiahnuť distribúciu cez Microsoft Store

7.2 Inštalácia WSL

Tento postup je v bash terminály. Nasledujúce skupiny programov budú inštalované pomocou APT managera:
.NET => `mono-devel msbuild dotnet-sdk-2.2`
Emscripten SDK => `libc6, ninja-build, node`
Ostatné => `python, powershell, apt-transport-https`

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys
↪  3FA7E0328081BFF6A14DA29AA6A19B38D3D831EF
$ echo "deb https://download.mono-project.com/repo/ubuntu stable-bionic main" |
↪  sudo tee /etc/apt/sources.list.d/mono-official-stable.list
$ sudo apt update
$ sudo apt install mono-devel msbuild -y
$ wget -q
↪  https://packages.microsoft.com/config/ubuntu/18.04/packages-microsoft-prod.deb
$ sudo dpkg -i packages-microsoft-prod.deb
$ sudo add-apt-repository universe
$ sudo apt update
$ sudo apt install apt-transport-https dotnet-sdk-2.2  -y
```

Výpis kódu 7.4: *Stiahnutie a inštalácia .NET závislostí*

Skript 7.4 nainštaluje potrebné .NET závislosti. Jedná sa o Mono SDK pomocou ktorého budeme robiť IL stripping dll súborov, .NET Core SDK pomocou ktorého budeme kompilovať C# kód pod .NET Standard compilation target. Balíček msbuild slúži pre prevolanie správneho SDK pri kompilácii C# projektu (Mono alebo .NET Core).

```
$ sudo apt install libc6 ninja-build nodejs -y
```

Výpis kódu 7.5: *Stiahnutie a inštalácia Emscripten SDK závislostí*

Balíčky z 7.5 slúžia pre Emscripten SDK. Balíček libc6 je niečo ako BCL pre .NET a obsahuje základné typy a funkcie pre programy napísané v jazyku C. Node.js je JavaScripový runtime (používa V8 engine) ktorý unožuje spúšťať JavaScript mimo prostredie prehliadača. Ninja-build je nízkoúrovňový build systém používaný pre AOT kompiláciu C# .

```
$ sudo apt install python powershell -y
```

Výpis kódu 7.6: *Stiahnutie a inštalácia Python a PowerShell*

Python je potrebné nainštalovať pretože vo WSL nie je nainštalovaný a používa ho Emscripten SDK. PowerShell je moja osobná preferencia a slúži na spustenie build taskov vo WSL (v cli je možné voalť PowerShell pomocou `pwsh` príkazu). Inštalácia umožňuje unifikovať skriptovací jazyk medzi OS (Linux a Windows).

Nainštalovanie prostredia a všetkých prerekvizít trvá približne 30 minút.[4]. Pre zjednodušenie inštalácie som pripravil shell skript `install_wsl_prerequisites.sh` ktorý sa nachádza v prílohe práce.Script obsahuje všetky predchádzajúce príkazy pre inštaláciu wsl prerekvizít.

7.2.1 Emscripten SDK

```
$ git clone https://github.com/emscripten-core/emsdk sdks/emsdk
$ cd sdks/emsdk
$ ./emsdk install sdk-1.38.28-64bit
$ ./emsdk activate sdk-1.38.28-64bit
$ ./emsdk_env.sh
```

Výpis kódu 7.7: *Naklonovanie repozitára s Emscripten SDK*

Emscripten SDK je potrebné stiahnuť z github repozitára. Prvý príkaz stiahne repozitár a následne je nutné nainštalovať sdk so správnym patchom (1.38.28). Po inštalácií je nutné SDK aktivovať. Po aktivácii je SDK pripravené na použitie. Pred každým použitím SDK je však nutné pridať environment variable do terminálu z ktorého budeme spúšťať kompiláciu projektu.

7.3 Voliteľné prerekvizity pre OS Windows

Príklady je možné spustiť pomocou WSL ale pre pohodlnejšie testovanie odporúčam použiť OS Windows a IDE Visual Studio[5].

Jednotlivé odporúčané časti pre OS Windows sú popísané v nasledujúcich podkapitolách. Pre jednoduchú inštaláciu odporúčam do OS Windows nainštalovať balíčkovací manager **Choco** `https://chocolatey.org/`.

Tak ako pri WSL som pripravil skript pre inštaláciu všetkých potrebných programov `install_windows_prerequisites.ps1`.

[4]Tento čas bude pri testovaní s WSL 2 pravdepodobne rádovo nižší `https://devblogs.microsoft.com/commandline/announcing-wsl-2/`

[5]Multiplatformná alternatíva je Visual Studio Code `https://code.visualstudio.com/`

7.3.1 .NET Core Sdk

Pre spustenie je nutné mať nainštalovaný .NET Core SDK vo verzii >=2.2.4. (.NET Core SDK nie je medzi WSL a OS Windows zdieľaná).

```
PS> choco install dotnetcore --version 2.2.4 -y
```

Výpis kódu 7.8: *Inštalácia .NET Core SDK pomocou choco*

Alternatívne je možné SDK stiahnuť ako klasický inštalátor zo stránky `https://dotnet.microsoft.com/download/thank-you/dotnet-sdk-2.2.203-windows-x64-installer`.

7.3.2 Node.js

```
PS> choco install nodejs --version 12.1.0 -y
```

Výpis kódu 7.9: *Inštalácia Node.js pomocou choco*

Alternatívne je možné Node.js stiahnuť ako klasický inštalátor zo stránky `https://nodejs.org/dist/v10.15.3/node-v10.15.3-x64.msi`.

7.3.3 Visual Studio

Pre pohodlnejšie testovanie odporúčam použiť OS Windows a ingrované developerské prostredie (IDE) Visual Studio.

```
PS> choco install visualstudio2019community -y
PS> choco install visualstudio2019-workload-netcoretools -y
PS> choco install visualstudio2019-workload-netcrossplat -y
PS> choco install visualstudio2019-workload-netweb -y
PS> choco install visualstudio2019-workload-webcrossplat -y
```

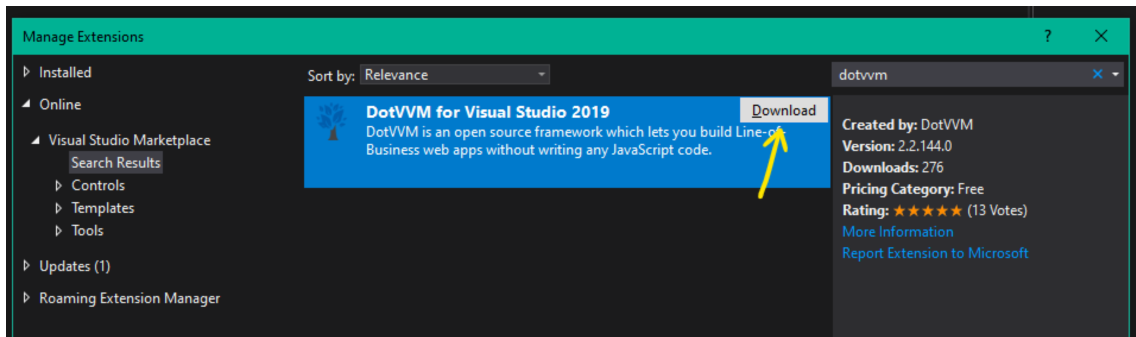
Výpis kódu 7.10: *Inštalácia Visual Studio a workloads pomocou choco*

Alternatívne je možné stiahnuť Visual Studio z `https://visualstudio.microsoft.com/`

DotVVM rozšírenie

Pre podporu frameworku DotVVM v Visual Studio je potrebné stiahnuť rozšírenie. Rozšírenie je možné pridať aj pomocou GUI v menu Visual Studio.
`Extensions/Manage Extensions`

Pre dokončenie inštalácie rozšírenia je potrebné vypnúť Visual Studio. Po vypnutí sa spustí inštalátor rozšírenia. Po nainštalovaní rozšírenia je možné spustiť projekt



Obr. 7.1: *Rozšírenie DotVMM vo Visual Studiu*

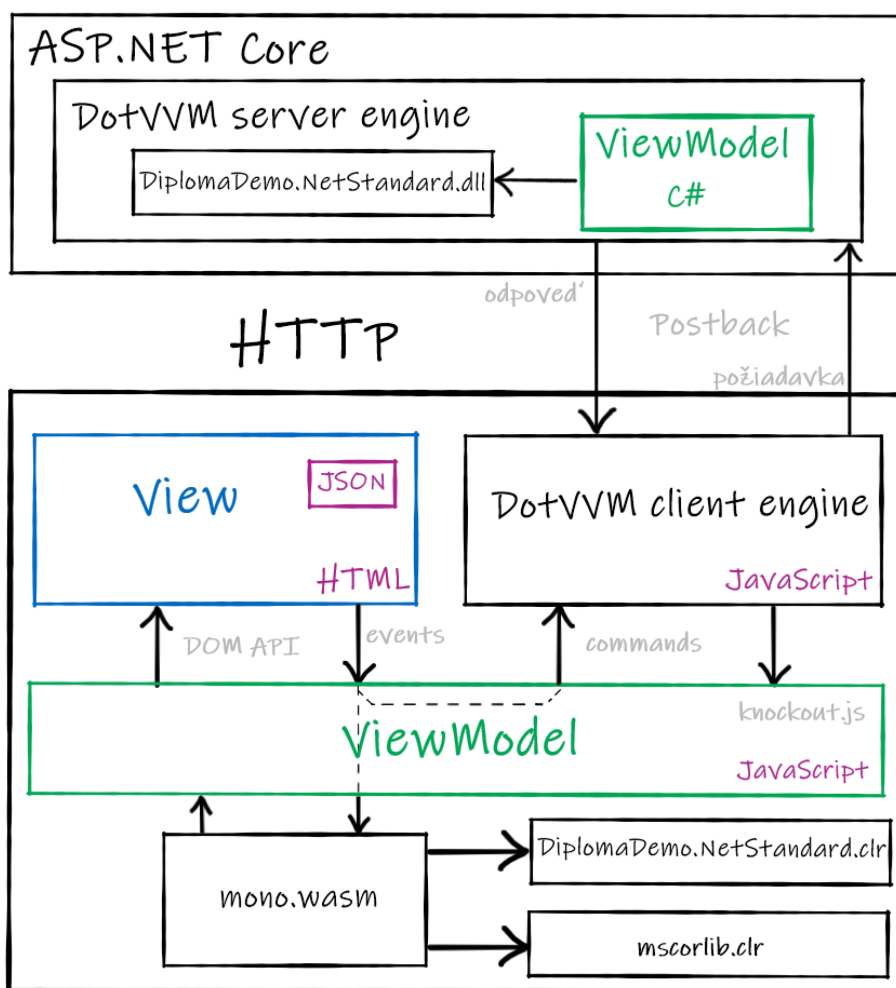
v ktorom sa používa DotVVM framework. Podrobný postup je možné nájsť na adrese `https://www.dotvvm.com/landing/dotvvm-for-visual-studio-extension`.

8 Implemetácia

Riešenie mojej práce má minimalizovať počet PostBackov z klienta na server, ktoré sú potrebné pre beh aplikácie. Po preskúmaní frameworku Blazor som sa rozhodol z neho použiť iba časť. Táto časť je .NET runtime Mono ktorý je vo formáte WebAssembly.

8.1 Konceptuálne riešenie zadania

Obmedzenie počtu PostBackov zo strany klienta na server bude umožnené vďaka zdieľaniu IL kódu medzi serverom a klientom (prehliadačom).



Obr. 8.1: *Principiálna schéma implementácie DotVVM a mono.wasm*

Na schéme 8.1 je konceptuálne riešenie zadania. Kód, ktorý bolo nutné spúšťať na serveri sa nachádza v `DiplomaDemo.NetStandard.dll`. Tento IL kód pomocou AOT kompilácie skompilujem do `DiplomaDemo.NetStandard.clr`. IL kód

z `DiplomaDemo.NetStandard.clr` bude následne vykonávaný pomocou mono.wasm na klientovi - v prehliadači.

Moje riešenie **neumožňuje** záložnú možnosť pre prípad, že webový prehliadač nebude technológiu WebAssembly podporovať z nasledujúcich dôvodov:

1 WebAssembly podporuje 85% prehliadačov na trhu[1]. WebAssembly je štandard W3C, preto ho podporujú všetky aktívne vyvíjané prehliadače.

2 DotVVM momentálne umožňuje s commands pracovať iba napriamo - cez DotVVM engine[2]. Pre riadnu implementáciu by bolo nutné vytvoriť v DotVVM plnohodnotný klientský ViewModel, ktorý by umožňoval vytvárať metódy a pracovať s hodnotami ViewModelu. Aktuálna implementácia totiž iba emuluje ViewModel a všetky commandy posiela na server.

8.2 Použité knižnice a tooling

Ako som spomínal v kapitole 7, pre určité kompilačné kroky sa využíva WSL. Nasledujúce kroky budú používať terminál PowerShell (nezaleží či v OS Windows alebo WSL).

Pre prácu s .NET projektmi som používal dotnet CLI, pre vytváranie kompilačných skriptov a bootraping projektov skriptovací jazyk PowerShell. Pre zjednotenie konfigurácie a vytvorenie jedného konfiguračného súboru som použil Node.js.

8.3 Štruktúra výsledného riešenia

Praktická implementácia má nasledovnú štruktúru projektov:

```
/
├── bootstraper/ .................... Zložka so skriptmi pre bootstrap dema
├── DiplomaDemo.DotVVM/ ............ Zložka projektu s DotVVM aplikáciou
├── DiplomaDemo.NetStandard/ ......... Zložka projektu so zdieľaným kódom
├── DiplomaDemo.sln ......................... Visual Studio solution súbor
└── bootstrap.ps1 ... Skript pre spustenie AOT kompilácie a bootstraperu
```

8.3.1 DiplomaDemo.NetStandard

Prvá zložka s projektom **DiplomaDemo.NetStandard** obsahuje kód ktorý bude zdieľaný medzi serverovým DotVVM ViewModelom a klientským ViewModelom. Zdieľanie kódu je umožnené pomocou compilation target .NET Standard.

[1] zdroj: `https://caniuse.com/#feat=wasm`

[2] Existuje možnosť PostBack handlerov, tá je však spojená s užívateľskými kontrolkammi a nie klientským ViewModelom

Pre AOT kompiláciu C# kódu do IL a následne do clr formátu používam OSS knižnicu **Uno.Wasm.Bootstrap**. Repozitár knižnice sa nachádza na githube na `https://github.com/nventive/Uno.Wasm.Bootstrap`.

Knižnica je distribuovaná pomocou .NET balíčkovacieho managera nuget. Pre správne fungovanie AOT kompilácie C# do WASM bolo nutné do projektu pridať balíčky **Uno.Wasm.Bootstrap** a **Uno.Wasm.Bootstrap.Cli**.

```
PS> dotnet add package Uno.Wasm.Bootstrap --version 1.0.0-dev.260
PS> dotnet add package Uno.Wasm.Bootstrap.Cli --version 1.0.0-dev.260
```

Výpis kódu 8.1: *Pridanie nuget balíčkov pomocou dotnet CLI*

Výsledná štruktúra projektu, ktorý umožňuje AOT kompiláciu je nasledovná:

```
DiplomaDemo.NetStandard/
├── Properties/
│   └── launchSettings.json ........ Konfigurácia pre ASP.NET Core
├── build.ps1 ...................... Skript ktorý spúšťa AOT build
├── build.sh ....................... Skript ktorý spúšťa AOT build
├── buildConfig.json ............... Konfigurácia pre AOT build
├── DiplomaDemo.NetStandard.csproj . Projekt pre referencovanie
├── DiplomaDemo.WasmAOT.csproj ..... Projekt pre AOT build
├── InitClass.cs ................... Trieda so statickou metódou Main
├── LinkerConfig.xml ............... Deskriptor Mono IL linkeru
└── TestClass.cs ................... Trieda so zdielaným kódom (metódami)
```

Kvôli problémom[3] s konfiguráciou pri kompilácii cez Visual Studio sú vytvorené 2 projekty.

1. Projekt **DiplomaDemo.NetStandard.csproj** - obsahuje šablónu projektu `Microsoft.NET.Sdk`[4] a compilation target na `netstandard2.0`.
2. Projekt **DiplomaDemo.WasmAOT.csproj** - používa šablónu projektu `Microsoft.NET.Sdk.Web`[5] a compilation target na `netstandard2.0`. Obsahuje tiež potrebné nastavenia premenných pre AOT kompiláciu IL kódu a vyššie spomínané balíčky `Uno.Wasm.Bootstrap` a `Uno.Wasm.Bootstrap.Cli`

AOT kompilácia využíva Mono IL linker na odstránenie prebytočného kódu (tried, metód a vlastností). Táto technika sa nazýva stripping. Pre správnu konfiguráciu

[3] Každá kompilácia projektu s aplikáciou (DotVVM) totiž kompilovala aj tento projekt a AOT kompiklácia trvá približne 3 minúty

[4] Šablóna projektu umožňuje používať východzie hodnoty premenných na základe ktorých sa tvorí kompikácia. Obsahuje napríklad wildcard pre zahrnutie všetkých *.cs súborov

[5] Šablóna je rozdielna oproti predchádzajúcemu projektu

Mono IL linkera je potrebné pridať deskriptor. Konfigurácia musí obsahovať názov assembly, ktorú chceme **vyradiť** zo strippingu.

Zložka obsahuje tiež skripty pre spustenie AOT buildu, ktoré sú spravené tak, že ich je možné spúšťať z OS Windows (build.ps1) alebo z WSL (build.ps1 alebo build.sh). Pre správne fungovanie skriptov je nutné nastaviť v konfiguračnom súbore **buildConfig.json** cestu k root zložke s Emscripten SDK. Skript po spustení pridá všetky premenné, ktoré sú potrebné pre AOT build.

8.3.2 DiplomaDemo.DotVVM

Zložka **DiplomaDemo.DotVVM** obsahuje projekt, ktorý používa DotVVM framework. Táto aplikácia slúži na demonštráciu AOT kompilácie a minimalizáciu Postbackov zo strany klienta na server.

```
DiplomaDemo.DotVVM/
├── Pages/
│   ├── Default.dothtml ................ View demo aplikácie
│   └── DefaultViewModel.cs ............ ViewModel demo aplikácie
├── Properties/
│   └── launchSettings.json ............ Konfigurácia pre ASP.NET Core
├── wwwroot/ ........................... Zložka Web root
│   ├── css/ ........................... Zložka s kaskádovými štýlmi aplikácie
│   ├── img/ ........................... Zložka s obrázkami aplikácie
│   ├── managed/ ....................... Zložka s clr súbormi
│   │   ├── DiplomaDemo.NetStandard.clr  Knižnica skompilovaná AOT
│   │   ├── mscorlib.clr ............... Microsoft Common Object Runtime Library
│   │   └── *.clr ...................... Ostanté clr súbory
│   ├── bootstrap.js ................... Bootstrapovací skript aplikácie
│   ├── config.js ...................... Konfiguračný súbor pre bootstrap.js
│   ├── mono.js ........................ Inicializačný súbor pre mono.wasm
│   ├── mono.wasm ...................... CLI runtime Mono
│   └── require.js ..................... Skript pre sťahovanie závislostí zo servera
├── DiplomaDemo.DotVVM.csproj .......... Projektový súbor
├── DotvvmStartup.cs ................... Konfigurácia frameworku DotVVM
├── Program.cs ......................... Vstupný súbor ASP.NET Core aplikácie
└── Startup.cs ......................... Konfigurácia ASP.NET Core aplikácie
```

Ukážka zdieľania kódu medzi serverom a klientom je v projekte **DiplomaDemo.DotVVM.csproj** referencovaný projekt **DiplomaDemo.NetStandard.csproj**. Trieda `TestClass` z referencovaného projektu je volaná vo ViewModeli (DefaultViewModel) DotVVM Projektu.

Výsledné súbory AOT kompilácie **DiplomaDemo.WasmAOT.csproj**[6] sú pridané do zložky **wwwroot** v rámci zjednodušenia spustenia demo aplikácie.

[6]Projekt obsahuje rovanké súbory ako DiplomaDemo.NetStandard.csproj

V aplikácii je povolená obsluha statických súborov neznámeho typu. Toto nastavenie je pre ASP.NET Core, čiže sa nachádza v súbore **Startup.cs**. Po tomto nastavení dokáže ASP.NET Core obsluhovať požiadavky na *.clr súbory, ktoré sa nachádzajú v zložke **wwwroot/managed/*.clr**.

Pre to, aby sme mohli pracovať s **mono.wasm**, je potrebné do HTML dokumentu vložiť nasledujúce súbory v tomto poradí[7]

1. **config.js** - tento skript obsahuje konfiguráciu potrebnú pre mono.wasm a bootstrap.js (aby sme vedeli ktoré závislosti sú potrebné zo servera stiahnuť - jedná sa o *.clr a *.wasm súbory)
2. **require.js** - OSS module a file loader, ktorý je určený pre použitie v prehliadači. Tento skript sa stará o volanie servera pre potrebné súbory a moduly.
3. **bootstrap.js** - skript ktorý načíta config.js, v ktorom sú súbory potrebné pre beh mono.wasm a knižnice `DiplomaDemo.NetStandard.clr`. Tento skript obsahuje callbacky, ktoré následne prevoláva skript mono.js po úspešnom načítaní runtime mono.wasm.
4. **mono.js** - skript načítava zo servera mono.wasm pomocou require.js. Stará sa tiež o prevolanie callbackov do bootstrap.js.

DotVVM umožňuje so závislosťami na stránke pracovať pomocou RequiredResource, ktoré zabezpečujú, že sa závislosti na stránke načítajú v správnom poradí - nie je teda potrebné ručne pridávať tieto závislosti do `head` elementu HTML dokumentu.

Pre nastavenie jednotlivých závislostí ich je nutné zaregistrovať do konfiguračného súboru **DotvvmStartup.cs**

```
28  config.Resources.Register("config", new ScriptResource()
29  {
30      Location = new UrlResourceLocation("~/config.js"),
31  });
32
33  config.Resources.Register("mono", new ScriptResource()
34  {
35      Location = new UrlResourceLocation("~/mono.js"),
36      Dependencies = new[] { "config",  "require", "wasm-bootstrap"}
37  });
```

Výpis kódu 8.2: *Ukážka pridania závislostí z DotvvmStartup.cs*

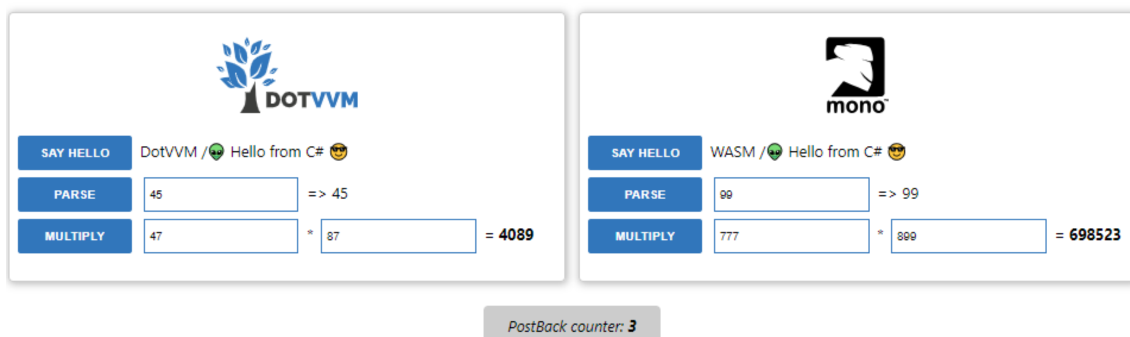
Ako je vidieť z výpisku 8.2, jednotlivé závislosti je možné medzi sebou reťaziť a DotVVM poskladá následne tieto závislosti do `head` elementu stránky v správnom poradí.

[7]Na poradí záleží, pretože skripty sa načítavajú sekvenčne

Aplikácia obsahuje jednu stránku ktorá sa skladá z View a ViewModelu[8]:

- View - je súbor **Default.dothtml**, ktorý obsahuje anotáciu v ktorej je referencia na ViewModel.
- ViewModel - je súbor **DefaultViewModel.cs** ktorý obsahuje metódy ktoré sú prevolávaná z View na klientovi. Práve funkcionalita ViewModelu - metódy a ich presun na stranu klienta sú gro mojej práce.

Na obrázku 8.2 je ukážka demo aplikácie. Táto aplikácia zdieľa rovnaký kód ktorý používa serverový ViewModel a klientský ViewModel. V ľavej časti sa nachádzajú tlačidlá, ktoré volajú cez PostBack metódy serverového ViewModelu. V pravej časti sa nachádzajú metódy, ktoré pomocou WebAssembly (mono.wasm) prevolávajú kód ktorý bol AOT kompiláciou skompilovaný do .clr formátu.



Obr. 8.2: *Ukážka demo aplikácie*

Aplikácia v spodnej časti obsahuje počítač PostBackov ktoré spraví DotVVM pri práci s demo aplikáciou. Pri testovaní aplikácie sa pre klientsku a serverovú časť používa ten istý kód ktorý je v zložke **DiplomaDemo.NetStandard**.

8.3.3 Bootstraper

Ako som spomínal kapitole 8.3.1, pre AOT kompiláciu sa používa nuget balíček **Uno.Wasm.Bootstrap**. Výsledok tejto kompilácie však nie je možné priamo použiť v našej aplikácii a preto som vytvoril sadu skriptov ktoré umožňujú transformovať výstup AOT kompilácie do štruktúry ktorá je vhodná pre použitie v projekte **DiplomaDemo.DotVVM**.

```
bootstraper/
├─ bootstrap.js ........................ Bootstrapovací skript aplikácie
├─ bootstraper.ps1 ..................................... Hlavný skript
├─ bootstraperConfig.json. .............. Konfigurácia pre boostraper.ps1
└─ createConfigJson.js...Skript pre vytvorenie konfiguračných súborov
```

[8]MVVM je popísaný v kapitole 6.2

9 Záver

V mojej práci som sa zaoberal webovým štandardom WebAssembly a frameworkom Blazor. Mojou úlohou bolo použiť tieto technológie pre minimalizáciu počtu PostBackov zo strany klienta, čo sa mi podarilo.

V teoretickej časti som zhrnul základy webových technológii a snažil som sa vysvetliť, prečo evolúcia smeruje k technológiám ako sú WebAssembly a Blazor. V praktickej časti som vytvoril toolchain pre zjednodušenie používania AOT kompilácie IL kódu do WebAssembly a jednoduché demo, ktoré demonštruje zdieľanie kódu medzi serverom a klientom (prehliadačom). Týmto zdieľaním je možné minimalizovať počet PostBackov na server, pretože je možné tento kód spustiť na klientovi, v prehliadači.

Technológia WebAssembly je revolučný koncept, pretože umožňuje spúšťať kód v sandboxe a má takmer natívnu rýchlosť. WebAssembly tiež prináša diverzifikáciu technológií, ktoré je možné používať v prehliadači a zároveň umožnila kooperáciu týchto technológií a JavaScriptu. Podľa môjho názoru tiež stojíme na hrane technologickej zmeny, pretože pre kód, ktorý budeme vytvárať, budeme používať compilation target WebAssembly. Tento compilation target umožní multiplatformné zdieľanie kódu bez nutnosti kompilácie pre špecifickú platformu (x86, AMD64, ARM, atď.).

Framework DotVVM umožňuje pohodlný vývoj webových aplikácií bez nutnosti písať boilerplate JavaScriptu, pri jeho používaní je však nutná častá komunikácia so serverom pomocou PostBackov. Súčasný trend vývoja webových aplikácií je mať aplikácie čo najviac „serverless“, čo znamená, že interakcia so serverom by mala byť minimálna. Tento trend je v kontraste so súčasným fungovaním DotVVM, a preto som sa rozhodol hľadať prístupy, ako túto interakciu klienta so serverom čo najviac minimalizovať. Zároveň som chcel zachovať filozofiu DotVVM, ktorou je nepísanie boilerplate JavaScriptu.

Záložnú možnosť pre prípad, že prehliadač nebude podporovať WebAssembly som nesplnil, pretože WebAssebmly je štandard W3C a podporuje ho takmer 86%[1] používaných prehliadačov. Zvyšných 14% sú nevyvíjanie prehliadače ako Internet Explorer, ktorých percento používania sa stále znižuje a využívajú zväčša iba na intranetové účely.

Myslím si, že som mojou prácou poukázal na ďalšiu možnosť pre budúcnosť frameworku DotVVM.

[1] Stav k 14.5. 2019, zdroj: `https://caniuse.com/#feat=wasm`

Literatúra

[1] Andreessen M. *proposed new tag: IMG* [online].[cit. 10. 12. 2018] Dostupné z URL: <`http://1997.webhistory.org/www.lists/www-talk.1993q1/0182.html`>.

[2] Berners-Lee T. *Tim Berners-Lee - Biography* [online].[cit. 10. 12. 2018] Dostupné z URL:<`https://www.w3.org/People/Berners-Lee/`>.

[3] Ambra J.*What is the technical history of web development?* [online].[cit. 10. 12. 2018] Dostupné z URL: <`https://www.quora.com/What-is-the-technical-history-of-web-development`>.

[4] Andreessen M. *INNOVATORS OF THE NET: BRENDAN EICH AND JAVASCRIPT* [online].[cit. 10. 12. 2018] Dostupné z URL: <`https://web.archive.org/web/20080208124612/http://wp.netscape.com/comprod/columns/techvision/innovators_be.html`>.

[5] Atkins Tab J. *THERE IS NO SUCH THING AS CSS4* [online].[cit. 10. 12. 2018] Dostupné z URL: <`https://www.xanthir.com/b4Ko0`>.

[6] Beattie A. *What were the "browser wars"?* [online].[cit. 10. 12. 2018] Dostupné z URL: <`https://www.investopedia.com/ask/answers/09/browser-wars-netscape-internet-explorer.asp`>.

[7] Belfiore J. *Microsoft Edge: Making the web better through more open source collaboration* [online].[cit. 10. 12. 2018] Dostupné z URL: <`https://blogs.windows.com/windowsexperience/2018/12/06/microsoft-edge-making-the-web-better-through-more-open-source-collaboration/#GmSJg4uFjBM5y8Hz.97`>.

[8] Belfiore J. *Microsoft Edge: Making the web better through more open source collaboration* [online].[cit. 19. 4. 2019] Dostupné z URL: <`https://blogs.windows.com/windowsexperience/2018/12/06/microsoft-edge-making-the-web-better-through-more-open-source-collaboration/`>.

[9] Berners-Lee T. *Re: status. Re: X11 BROWSER for WWW* [online].[cit. 10. 12. 2018] Dostupné z URL: <`http://lists.w3.org/Archives/Public/www-talk/1991SepOct/0003.html`>.

[10] Bert B. *A brief history of CSS until 2016* [online].[cit. 10. 12. 2018] Dostupné z URL: <`https://www.w3.org/Style/CSS20/history.html`>.

[11] Brainvire *Six Benefits of Using MVC Model for Effective Web Application Development* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://www.brainvire.com/six-benefits-of-using-mvc-model-for-effective-web-application/development/>.

[12] Brendan E. *záznam z Fin.JS - How was JavaScript invented?* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://www.youtube.com/watch?v=XOmhtfTrRxc>.

[13] Bright P. *The Web is getting its bytecode: WebAssembly* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://arstechnica%.com/information-technology/2015/06/the-web-is-getting-its-bytecode-webassembly/>.

[14] Bynens M. *záznam z konferencie SmashingConf London 2018 - JavaScript Engine Internals for JavaScript Developers* [online].[cit. 2. 5. 2019] Dostupné z URL: <https://www.youtube.com/watch?v=-lt6a9kbc_k>.

[15] Chishkala I. *The HTTP/2 Protocol: Its Pros & Cons and How to Start Using It* [online].[cit. 19. 4. 2019] Dostupn? z URL: <https://www.upwork.com/hiring/development/the-http2-protocol-its-pros-cons-and-how-to-start-using-it/>.

[16] Claburn T. *WebAssembly fandom kills Google's Portable Native Client* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://www.theregister.co.uk/2017/05/31/googles_portable_native_client_to_be_killed/>.

[17] Clark L. *A cartoon intro to WebAssembly* [online].[cit. 2. 5. 2019] Dostupné z URL: <https://hacks.mozilla.org/2017/02/a-cartoon-intro-to-webassembly/>.

[18] Clark L. *A crash course in just-in-time (JIT) compilers* [online].[cit. 2. 5. 2019] Dostupné z URL: <https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/>.

[19] Clark L. *Making WebAssembly even faster: Firefox's new streaming and tiering compiler* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://hacks.mozilla.org/2018/01/making-webassembly-even-faster-firefoxs-new-streaming-and-tiering-compiler/>.

[20] Clark L. *What makes WebAssembly fast?* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/>.

[21] Clark L. *Where is WebAssembly now and what's next?* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://hacks.mozilla.org/2017/02/where-is-webassembly-now-and-whats-next/>.

[22] Clark L. *záznam z konferencie CovalenceConf 2019 - WebAssembly's Post-MVP Future* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://www.youtube.com/watch?v=9wDx4RuyIMU>.

[23] Clark L., Schneidereit T., Wagner L. *WebAssembly's post-MVP future: A cartoon skill tree* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://hacks.mozilla.org/2018/10/webassemblys-post-mvp-future/>.

[24] Cleverism *Active Server Pages (ASP)* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://www.cleverism.com/skills-and-tools/active-server-pages-asp/>.

[25] Cooper S. *Whatever happened to Netscape?* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://www.engadget.com/2014/05/10/history-of-netscape/?guccounter=1>.

[26] De Icaza M. *Hello WebAssembly* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://www.mono-project.com/news/2017/08/09/hello-webassembly/>.

[27] developers.google *Progressive Web Apps* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://developers.google.com/web/progressive-web-apps/>.

[28] DotVVM dokumentácia *DotVVM dokumentácia* [online].[cit. 10. 5. 2019] Dostupné z URL: <https://www.dotvvm.com/docs/latest>.

[29] Dr. Rauschmayer A. *Running code fast in web browsers: PNaCl versus asm.js* [online].[cit. 19. 4. 2019] Dostupné z URL: <http://2ality.com/2013/06/pnacl-vs-asmjs.html>.

[30] Dutton S. *záznam z konferencie GDD Europe '17 - Progressive Web Apps: What, Why, and How?* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://www.youtube.com/watch?v=eodArdGRIVQ>.

[31] Eberhardt C.*záznam z JS Monthly London Meetup - WebAssembly and the Death of JavaScript* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://www.youtube.com/watch?v=3LWgbjVWLug>.

[32] Factory.hr *HTTP/2: the difference between HTTP/1.1, benefits and how to use it* [online].[cit. 19. 4. 2019] Dostupn? z URL: <https:

//medium.com/@factoryhr/http-2-the-difference-between-http-1-1-benefits-and-how-to-use-it-38094fa0e95b>.

[33] Fraternali P.. *Common Gateway Interface* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://www.slideshare.net/fraterna/common-gateway-interface-17405944>.

[34] Gal A. *CHROME WON* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://andreasgal.com/2017/05/25/chrome-won/>.

[35] Gamage Ashen T. *Evolution of HTTP - HTTP/0.9, HTTP/1.0, HTTP/1.1, Keep-Alive, Upgrade, and HTTPS* [online].[cit. 19. 4. 2019] Dostupn? z URL: <https://medium.com/platform-engineer/evolution-of-http-69cfe6531ba0>.

[36] Garrett James J. *Ajax: A New Approach to Web Applications* [online].[cit. 10. 12. 2018] Dostupné z URL: <http://adaptivepath.org/ideas/ajax-new-approach-web-applications/>.

[37] Garsiel T. *How browsers work* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://taligarsiel.com/Projects/howbrowserswork1.htm>.

[38] Geek History *The 1990s browser wars Microsoft IE versus Netscape Navigator* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://web.archive.org/web/20170519195828/https://geekhistory.com/content/1990s-browser-wars-microsoft-ie-versus-netscape-navigator>.

[39] Greggman G. *Thoughts on asm.js vs PNaCl* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://games.greggman.com/game/thoughts-on-asm-js-vs-pnacl/>.

[40] Guthrie s. *ASP.NET MVC, Web API, Razor and Open Source* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://weblogs.asp.net/scottgu/asp-net-mvc-web-api-razor-and-open-source>.

[41] Harris M. *záznam z The JS Roundabout 2017 - PWAs vs Native (aka There's A Progressive Web App For That)* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://www.youtube.com/watch?v=vhg01Ml-8pI>.

[42] Heisler R. *A visual history of Netscape Navigator* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://www.networkworld.com/article/2833526/software/a-visual-history-of-netscape-navigator.html>.

[43] IETF HTTP Working Group *HTTP/2 Frequently Asked Questions* [online].[cit. 19. 4. 2019] Dostupn? z URL: <https://http2.github.io/faq/>.

[44] info.cern.ch *Restoring the first website* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://first-website.web.cern.ch/>.

[45] interactivepython.org. *Fundamentals of Web Programming* [online]. [cit. 10. 12. 2018] Dostupné z URL:<http://interactivepython.org/runestone/static/webfundamentals/WWW/history.html>.

[46] Ironman, Political Calculations *Here's Why The Dot Com Bubble Began And Why It Popped* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://www.businessinsider.com/heres-why-the-dot-com-bubble-began-and-why-it-popped-2010-12>.

[47] Jaggavarapu M. *Presentation Patterns : MVC, MVP, PM, MVVM* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://manojjaggavarapu.wordpress.com/2012/05/02/presentation-patterns-mvc-mvp-pm-mvvm/>.

[48] javatpoint *Model 1 and Model 2 (MVC) Architecture* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://www.javatpoint.com/model-1-and-model-2-mvc-architecture>.

[49] Lander R. *Introducing .NET 5* [online].[cit. 14. 5. 2019] Dostupné z URL: <https://devblogs.microsoft.com/dotnet/introducing-net-5/>.

[50] Larabel M. *Mozilla's Project Mortar Wants Pepper API Flash & PDFium In Firefox* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://www.phoronix.com/scan.php?page=news_item&px=Mozilla-Project-Mortar>.

[51] Lattner Ch. *Introduction to the LLVM - Compiler System* [online].[cit. 14. 5. 2019] Dostupné z URL: <http://llvm.org/pubs/2008-10-04-ACAT-LLVM-Intro.pdf>.

[52] lld.llvm.org *WebAssembly lld port* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://lld.llvm.org/WebAssembly.html>.

[53] Looper J. *A Guide to JavaScript Engines for Idiots* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://developer.telerik.com/featured/a-guide-to-javascript-engines-for-idiots/>.

[54] Mackie K. *Microsoft Edge Browser To Get New Rendering Engine but EdgeHTML Continues* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://redmondmag.com/articles/2018/12/10/edgehtml-continues.aspx>.

[55] Massi B. *Building an Open Source .NET Foundation* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://medium.com/microsoft-open-source-stories/building-an-open-source-net-foundation-2fa0fb117584>.

[56] mbasso *Awesome Wasm* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://github.com/mbasso/awesome-wasm>.

[57] MDN Web docs *Introduction to the DOM* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction>.

[58] MDN Web docs *Server-side web frameworks* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Web_frameworks>.

[59] MDN Web docs *Understanding WebAssembly text format* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format>.

[60] MDN Web docs *WebAssembly.Global* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/Global>.

[61] MDN Web docs *WebAssembly* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://developer.mozilla.org/en-US/docs/WebAssembly>.

[62] MERANT *Appendix A: Introduction to the World-Wide Web* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://supportline.microfocus.com/documentation/books/sx20books/piwebi.htm>.

[63] Microsoft Docs *.NET architectural components* [online].[cit. 2. 5. 2019] Dostupné z URL: <https://docs.microsoft.com/en-us/dotnet/standard/components>.

[64] Microsoft Docs *.NET Standard* [online].[cit. 2. 5. 2019] Dostupné z URL: <https://docs.microsoft.com/en-us/dotnet/standard/net-standard>.

[65] Microsoft docs *ASP.NET overview* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://docs.microsoft.com/en-us/aspnet/overview>.

[66] Mihajlija M. *WebAssembly: How and why* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://blog.logrocket.com/webassembly-how-and-why-559b7f96cd71>.

[67] MIT Education *The World Wide Web* [online]. [cit. 10. 12. 2018] Dostupné z URL: <`https://web.archive.org/web/20100608081841/http://web.mit.edu/invent/iow/berners-lee.html`>.

[68] Mono Docs *Oficiálna dokumentácia projektu Mono* [online].[cit. 2. 5. 2019] Dostupné z URL: <`https://www.mono-project.com/docs/`>.

[69] mono-project.com *About Mono* [online].[cit. 10. 12. 2018] Dostupné z URL: <`https://www.mono-project.com/docs/about-mono/`>.

[70] Mozilla Research *WebAssembly format* [online].[cit. 10. 12. 2018] Dostupné z URL: <`https://research.mozilla.org/webassembly/`>.

[71] MozillaWiki *Gecko:Overview* [online].[cit. 19. 4. 2019] Dostupné z URL: <`https://wiki.mozilla.org/Gecko:Overview`>.

[72] Nelson B. *LLVM on the Web - Using Portable Native Client to run Clang/LLVM in the Browser* [online].[cit. 19. 4. 2019] Dostupné z URL: <`https://llvm.org/devmtg/2015-02/slides/brad-pnacl.pdf`>.

[73] Netscape *Netscape and sun announce javascript, the open, cross-platform object scripting language for enterprise networks and the internet* [online].[cit. 10. 12. 2018] Dostupné z URL: <`https://web.archive.org/web/20080213171613/http://wp.netscape.com:80/newsref/pr/newsrelease67.html`>.

[74] Netscape Communications Corporation *THE SSL PROTOCOL* [online].[cit. 10. 12. 2018] Dostupné z URL: <`https://web.archive.org/web/19970614020952/http://home.netscape.com/newsref/std/SSL.html`>.

[75] Node.js Foundation *October Brings Node.js 10.x to LTS and Node.js 11 to Current!* [online].[cit. 19. 4. 2019] Dostupné z URL: <`https://medium.com/@nodejs/october-brings-node-js-10-x-to-lts-and-node-js-11-to-current-ae19f8f12b51`>.

[76] ongraph *What is MVC Architecture in a Web Based Application?* [online].[cit. 10. 12. 2018] Dostupné z URL: <`https://www.ongraph.com/what-is-mvc-architecture-in-a-web-based-application/`>.

[77] ORACLE *Server-Side JavaScript Guide* [online].[cit. 10. 12. 2018] Dostupné z URL: <`https://docs.oracle.com/cd/E19957-01/816-6411-10/contents.htm`>.

[78] OWASP Foundation *What are web applications?* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://www.owasp.org/index.php/What_are_web_applications%3F>.

[79] PEW RESEARCH CENTER *World Wide Web Timeline* [online].[cit. 10. 12. 2018] Dostupné z URL: <http://www.pewinternet.org/2014/03/11/world-wide-web-timeline/>.

[80] Peyrott S. *A Brief History of JavaScript* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://auth0.com/blog/a-brief-history-of-javascript/>.

[81] Pflug K. *Welcoming Progressive Web Apps to Microsoft Edge and Windows 10* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://blogs.windows.com/msedgedev/2018/02/06/welcoming-progressive-web-apps-edge-windows-10/>.

[82] PHP docs *History of PHP* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://www.php.net/manual/en/history.php.php>.

[83] Robinson D.,Coar K. *The Common Gateway Interface (CGI) Version 1.1* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://tools.ietf.org/html/rfc3875>.

[84] Roth D. *Blazor now in official preview!* [online].[cit. 7. 5. 2019] Dostupné z URL: <https://devblogs.microsoft.com/aspnet/blazor-now-in-official-preview/>.

[85] Russell A. *Progressive Web Apps: Escaping Tabs Without Losing Our Soul* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/>.

[86] Russell A. *What, Exactly, Makes Something A Progressive Web App?* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://infrequently.org/2016/09/what-exactly-makes-something-a-progressive-web-app/>.

[87] Sanderson S., Roth D., Latham L.*Introduction to Blazor* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://blazor.net/docs/introduction/index.html>.

[88] Sansonetti L. *Mono and WebAssembly - Updates on Static Compilation* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://www.mono-project.com/news/2018/01/16/mono-static-webassembly-compilation/>.

[89] Sherman P. *How Single-Page Applications Work* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://blog.pshrmn.com/entry/how-single-page-applications-work/>.

[90] Simmons D. *Get started with WebAssembly - using only 14 lines of JavaScript* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://medium.freecodecamp.org/get-started-with-webassembly-using-only-14-lines-of-javascript-b37b6aaca1e4>.

[91] Soroker T. *CLR vs JVM: How the Battle Between C# and Java Extends to the VM-Level* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://blog.takipi.com/clr-vs-jvm-how-the-battle-between-net-and-java-extends-to-the-vm-level/>.

[92] squeak.org. *MVC* [online].[cit. 10. 12. 2018] Dostupné z URL: <http://wiki.squeak.org/squeak/1767>.

[93] TECH.SAIGONIST.COM. *Assembly language vs Bytecode vs WebAssembly vs Asm.js* [online].[cit. 14. 5. 2019] Dostupné z URL: <https://tech.saigonist.com/b/code/assembly-language-vs-bytecode-vs-webassembly-vs-asmjs.html>.

[94] TutrialsPoint *JSON with Ajax* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://www.tutorialspoint.com/json/json_ajax_example.htm>.

[95] W3C *Current Members* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://www.w3.org/Consortium/Member/List>.

[96] W3C *HTML Tags* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Tags.html>.

[97] Wagner L. *asm.js AOT compilation and startup performance* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://blog.mozilla.org/luke/2014/01/14/asm-js-aot-compilation-and-startup-performance/>.

[98] Wagner L. *WebAssembly* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://blog.mozilla.org/luke/2015/06/17/webassembly/>.

[99] Wallker J. *Cheat Sheet: What you need to know about Edge on Chromium* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://www.onmsft.com/how-to/cheat-sheet-what-you-need-to-know-about-edge-on-chromium>.

[100] Wasson M. *ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://msdn.microsoft.com/en-us/magazine/dn463786.aspx>.

[101] web.stanford.edu *Introduction to JavaScript!* [online].[cit. 10. 12. 2018] Dostupné z URL: <web.stanford.edu/class/cs98si/slides/overview.html>.

[102] WebAssembly Community Group *Types* [online].[cit. 19. 4. 2019] Dostupné z URL: <https://webassembly.github.io/spec/core/syntax/types.html>.

[103] webassembly.org. *FAQ* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://webassembly.org/docs/faq/>.

[104] Wilson B. *Netscape Navigator* [online].[cit. 10. 12. 2018] Dostupné z URL: <http://www.blooberry.com/indexdot/history/netscape.htm>.

[105] Zakai A. *BIG WEB APP? COMPILE IT!* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://kripken.github.io/mloc_emscripten_talk/#/>.

[106] Zakai A. *EMSCRIPTEN & ASM.JS: C++'S ROLE IN THE MODERN WEB* [online].[cit. 10. 12. 2018] Dostupné z URL: <https://kripken.github.io/mloc_emscripten_talk/cppcon.html>.

Zoznam používaných skratiek

AJAX	Asynchronous JavaScript and XML
AOT	Ahead of Time
API	Application Programming Interface
ASP	Active Server Pages
CERN	Conseil Européen pour la recherche nucléaire
CSS	Cascade Style Sheets
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLI	Command Line Interface
CLR	Common Language Runtime
DOM	Domain Object Model
ECMA	European Computer Manufacturers Association
GC	Garbage Collector
GWT	Google Web Toolkit
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
IE	Internet Explorer
IL	Intermediate Language
JIT	Just In Time
JS	JavaScript
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LLVM	Low Level Virtual Machine
LLVM IR	Low Level Virtual Machine Intermediate Representation
MVP	Minimum Valuable Product
NaCl	Native Client
NPAPI	Native Pepper API
npm	Node.js package manager
OSS	Open Source Software
PNaCl	Portable Native Client
PPAPI	Portable Pepper API
PWA	Progressive Web Application
RISC	Reduced Instruction Set Computer
SDK	Software Development Kit
SPA	Single Page Application
SIMD	Single instruction, multiple data
OS	Operačný systém

OSS	Open Source Software
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UWP	Universal Windows Platform
W3C	World Wide Web Consortium
WASM	Webassembly
WHATWG	Web Hypertext Application Technology Working Group

Zoznam príloh

A Obsah prílohy **96**

A Obsah prílohy

```
diploma-attachments.zip
└── DiplomaDemo/
    ├── bootstraper/
    │   ├── bootstrap.js .................................... 10 KB
    │   ├── bootstraper.ps1 .................................. 3 KB
    │   ├── bootstraperConfig.json ........................... 1 KB
    │   └── createConfigJson.js .............................. 1 KB
    └── DiplomaDemo.DotVVM/
        ├── Pages/
        │   ├── Default.dothtml .............................. 4 KB
        │   └── DefaultViewModel.cs .......................... 1 KB
        ├── Properties/
        │   └── launchSettings.json .......................... 1 KB
        └── wwwroot/
            ├── css/
            │   └── styles.css ............................... 2 KB
            ├── img/
            │   ├── logo-dotvvm.svg ......................... 11 KB
            │   ├── Mono_project_logo.svg .................... 7 KB
            │   └── UTKO_color_RGB_EN.png ................... 34 KB
            └── managed/
                ├── aot-dummy.clr ............................ 3 KB
                ├── DiplomaDemo.NetStandard.clr .............. 5 KB
                ├── Mono.Security.clr ........................ 3 KB
                ├── mscorlib.clr ........................... 546 KB
                ├── netstandard.clr .......................... 3 KB
                ├── System.clr .............................. 46 KB
                ├── System.ComponentModel.Composition.clr .... 3 KB
                ├── System.Core.clr .......................... 3 KB
                ├── System.Data.clr .......................... 3 KB
                ├── System.Drawing.Common.clr ................ 3 KB
                ├── System.IO.Compression.clr ................ 3 KB
                ├── System.IO.Compression.FileSystem.clr ..... 3 KB
                ├── System.Memory.clr ........................ 3 KB
                ├── System.Net.Http.clr ..................... 44 KB
                ├── System.Numerics.clr ...................... 3 KB
                ├── System.Runtime.Serialization.clr ......... 3 KB
                ├── System.ServiceModel.Internals.clr ........ 3 KB
                ├── System.Transactions.clr .................. 3 KB
                ├── System.Web.Services.clr .................. 3 KB
                ├── System.Xml.clr ........................... 3 KB
                ├── System.Xml.Linq.clr ...................... 3 KB
                ├── WebAssembly.Bindings.clr ................ 19 KB
                └── WebAssembly.Net.Http.clr ................ 14 KB
```

```
diploma-attachments.zip
│   │   │   └── WebAssembly.Net.WebSockets.clr ........................ 16 KB
│   │   ├── bootstrap.js ............................................. 10 KB
│   │   ├── config.js ................................................. 6 KB
│   │   ├── mono.js ................................................. 643 KB
│   │   ├── mono.wasm .............................................. 5825 KB
│   │   └── require.js ............................................... 17 KB
│   ├── DiplomaDemo.DotVVM.csproj ...................................... 1 KB
│   ├── DotvvmStartup.cs ............................................... 2 KB
│   ├── Program.cs ..................................................... 1 KB
│   └── Startup.cs ..................................................... 1 KB
├── DiplomaDemo.NetStandard/
│   ├── Properties/
│   │   └── launchSettings.json ........................................ 1 KB
│   ├── build.ps1 ...................................................... 1 KB
│   ├── build.sh ....................................................... 1 KB
│   ├── buildConfig.json ............................................... 1 KB
│   ├── DiplomaDemo.NetStandard.csproj ................................. 1 KB
│   ├── DiplomaDemo.WasmAOT.csproj ..................................... 1 KB
│   ├── InitClass.cs ................................................... 1 KB
│   ├── LinkerConfig.xml ............................................... 1 KB
│   └── TestClass.cs ................................................... 1 KB
├── bootstrap.ps1 ...................................................... 1 KB
└── DiplomaDemo.sln .................................................... 3 KB
└── scripts/
    ├── install_windows_prerequisites.ps1 .............................. 1 KB
    └── install_wsl_prerequisites.sh ................................... 1 KB
```